Supporting Dependable Distributed Applications through a Component-Oriented Middleware-Based Group Service

Katia Saikoski and Geoff Coulson

Computing Dept.
Lancaster University
Lancaster LA1 4YR, UK
+44 1524 593054
ksaikoski@terra.com.br; geoff@comp.lancs.ac.uk
http://www.comp.lancs.ac.uk

Abstract. Dependable distributed applications require flexible infrastructure support for controlled redundancy, replication, and recovery of components and services. However, most group-based middleware platforms, which are increasingly being used as implementation environments for such systems, fail to provide adequate flexibility to meet diverse application requirements. This paper presents a group-based middleware platform that aims at maximal flexibility. In particular, flexibility is provided at design time, deployment time and run-time. At design and deployment time, the developer can configure a system by assembling software components shaped to a specific use. Then, at run-time, s/he can dynamically reconfigure the resulting system to adjust it to new circumstances, or can add arbitrary machinery to enable the system to perform self-adaptation. As examples, levels of fault tolerance can be dynamically increased and decreased as desired by adding, removing or replacing replicas; or the underlying communications topology can be adapted by switching from point-to-point TCP to multicast as numbers of replicants increase. Importantly, it is not necessary that the shape that run-time reconfiguration takes has been foreseen at design or deployment time. Our proposed solution employs software component technology and computational reflection as the basic means by which to perform and manage configuration and reconfiguration.

Keywords
Dependability, groups, components, reflection, middleware.

1 Introduction

The past few years have seen a significant increase in the importance of group-based distributed applications. Although they have in common the use of multiple end-points, the class of group applications is characterised by great diversity: while the support of *dependability* (e.g. fault-tolerance or high availability based

on replication of servers) is perhaps the largest subset, dissemination of audio and video, distribution of events, and computer supported cooperative work are also important constituents of this class. Ideally, both dependability scenarios and these other group-related areas would be supported by a common underlying distributed platform.

Unfortunately, present day distributed object based middleware platforms (e.g. the Common Object Oriented Architecture (CORBA) [1] or the The Distributed Component Object Model (DCOM) [2]), which are increasingly being used as implementation environments for such applications, fail to provide suitable support for group applications in their full generality. And where they do provide support it is typically targeted at a limited subset of group applications, or is piece-meal and non-integrated. Examples from the CORBA world are the fault tolerance service [3], and the (completely separate) CORBA event service [4]. This lack of integration is problematic in a number of ways. In particular, it leads to missed opportunities for common solutions (e.g. in multicast channels, group addressing, access control and management) and makes it harder than it should be to combine related services (e.g. to provide a fault tolerant event service).

The broad aim of the research described in this paper is to provide a fully general and integrated platform for the support of group applications. Because of the diversity involved, this is a challenging undertaking. The different types of group applications vary dramatically in their requirements in terms of topology, dynamicity of membership, authentication of membership, logging of messages, reliability of communications, ordering guarantees on message delivery, use of network multicast etc.

Our approach is to attempt to satisfy all such requirements in a middleware environment known as OpenORB [5, 6, 7] which, in addition to supporting (group) applications, also supports the development of generic group services. This is primarily achieved through the use of component technology [8] which allows programmers to build applications and generic group services in terms of an extensible library of basic building blocks (components). Components are used not only at the application/ services level; the middleware itself is built from components. This allows groups to be used recursively to support communication among the various middleware components themselves. As a simple example, basic middleware components such as the name service can be made fault tolerant by means of groups.

A further aspect of our work is the support of *run-time reconfiguration* of running group applications or services. Runtime reconfiguration is essential for those applications that need to cope with changing requirements; in particular, for systems that cannot be switched off easily (e.g., control systems), or systems that cannot be switched off for long periods (e.g., telecom systems). As an example, consider a highly available service built from a group of server replicas. In such a scenario, it may be desirable to change the inter-replica reliability protocol (e.g. from an active replica to a passive replica scheme) as the number of replicas, the number of clients, or the performance/ reliability needs change over

time. Another (non-dependability-related) example could involve adding or removing media filters such as video compressors to support media dissemination in multimedia conferences involving nodes with heterogeneous network connectivity and processing capability [9]. To support such requirements, we apply the notion of *reflection* [10]. For example, we maintain run-time component graphs, which allow component configurations to be adapted in a "direct manipulation" style (see Sect. 2).

This paper is structured as follows. First, Sect. 2 describes our basic middleware architecture and its reflective, component based, computational model. Next we discuss, in Sects. 3 and 4, our support for the construction and run-time reconfiguration of component-based groups (with a focus on dependability-related services). Section 5 then presents an example of the use of our group support services. Finally, Sect. 6 deals with related work and Sect. 7 offers concluding remarks.

2 Background on OpenORB

OpenORB [5, 6, 7] is a flexible middleware platform built according to a component-based architecture. It is loosely inspired by the ISO Reference Model for Open Distributed Processing (RM-ODP) [11]. At deployment-time, components are selected and appropriately composed to create a specific instance of the middleware. For example, components encapsulating threads, buffer management, the (Internet Inter-ORB Protocol) IIOP protocol and the Portable Object Adapter may be selected, among others, for placement within a CORBA address space. In addition, components can be loaded into address spaces (using RM-ODP terminology, we will refer to address spaces as *capsules*) at run-time, and there are no restrictions on the language in which they are written. Components can be either *primitive* or *composite*. The former have no internal structure (at least not at the level of the component model) whereas the latter are formed as a composition of nested components (either primitive or composite). As we explain below, reflection is used to facilitate the reconfiguration of the set of components configured into a composite component or a capsule, and thus dynamically adapt the middleware functionality at run-time.

In OpenORB, component interfaces are specified in (an extended version of) the CORBA Interface Definition Language (IDL), and components may export any number of interface types. Furthermore, new interface instances can be created on demand at run-time. Multiple interface types provides separation of concerns; for example, operations to control the transfer of state between replicas in a fault tolerant service can be separated from operations to provide the service itself. Multiple interface instances are then useful in giving multiple clients their own private "view" of the component.

As well as standard operational interactions (i.e. method calls), our extended IDL supports the *signal* and *stream* interaction types defined in RM-ODP. Signal interactions, which are one-way messages, are typically used for primitive events; and stream interactions, which additionally involve isochronous timing

properties, are used for audio or video etc. Along with the authors of RM-ODP, we claim that this set of interaction types is canonical and functionally complete, or at least can serve to underpin any other conceivable interaction type. Apart from interaction types, each interface takes one of two possible roles: either *provided* or *required*. Provided interfaces represent the services offered to the component's environment, while required interfaces represent the services the component needs from its environment (in terms of the provided interfaces of other components). This explicit statement of dependency eases the task of composing components in the first place and also makes it feasible to replace components in running configurations (see below).

Communication between the interfaces of different components can only take place if the interfaces have been *bound*. In terms of role, required interfaces can only be bound to provided interfaces and vice versa. To-be-bound interfaces must also match in terms of their interaction types (i.e. method signatures etc.). There are two categories of bindings: local bindings and distributed bindings (see Fig. 1). The former, which are simple and primitive in nature, can be used only where the to-be-bound interfaces reside in the same capsule; they effectively terminate the recursion implicit in the fact that distributed bindings have interfaces which need to be bound to the interface they are binding! Distributed bindings are composite and distributed components which may span capsule or machine boundaries. Internally, these bindings are composed of sub-components (themselves bound by means of local bindings) that wrap primitive facilities of the underlying system software. Examples are primitive network-level connections (e.g., a "multicast IP binding"), media filters, stubs, skeletons etc. Distributed bindings are often constructed in a hierarchical (nested) manner; for example a "reliable multicast binding" may be created by encapsulating a "primitive" multicast IP binding and components that offer SRM-like functionality [12].

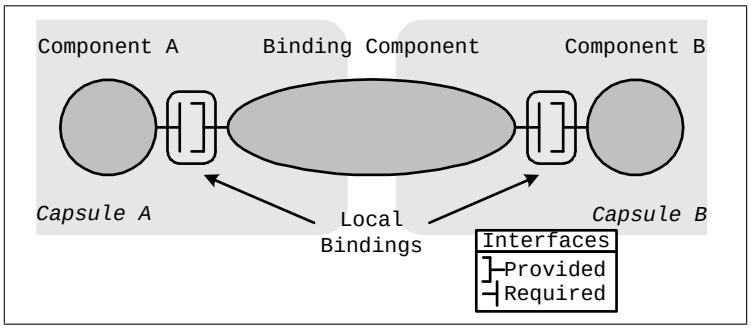

Fig. 1. Bindings supporting local and distributed interaction between components

It should be noted that distributed bindings are not unique in being composite and distributed components; any OpenORB component is allowed to have

these properties. In other words, distributed binding components do not enjoy any particular status as compared to other components.

Components are created by *factory* components (or simply "factories"). Factories for composite components are implemented as compositions of factories, each of which deals with the composite component's various constituents. Often, factories are parameterised with *templates* which define the required configuration and characteristics of to-be-created components. Templates are specified in the Extensible Markup Language (XML) [13]; specifically group-oriented templates are discussed in detail below.

In terms of *reflection*, every OpenORB component has an associated "meta-space" that is accessible from any of the component's interfaces, and which provides reflective access to, and control of, the component in various ways. To help separate concerns, meta-space is partitioned into various orthogonal "meta-models". For example, the *interface meta-model* allows access to a component's interface signatures and bindings, and the *resources meta-model* gives access to the resources (e.g. threads or memory) used by the component. The meta-model of most relevance to this paper is the *compositional meta-model*. This takes the form of a graph data structure which serves as a causally-connected self-representation of a capsule's internal topological structure (in terms of component composition). This means that manipulations of the graph's topology result in corresponding changes in the capsule's internal topology (and vice versa). Support is provided to make such changes atomic (e.g. by freezing any threads executing in the changed component). The other meta-models, which are beyond the scope of this paper, are described in [6]. More generally, OpenORB itself is discussed in more detail in [7].

3 An Overview of GOORB

GOORB (*Group support for OpenORB*) is a flexible OpenORB-based infrastructure for the creation and maintenance of both *group types* and *group instances*. At design and start up time, group types are defined based on templates, and on the re-use and assembly of software components from an extensible library. Then, at run-time, group instances are created on the basis of group type specifications and can subsequently be flexibly reconfigured to adjust them to dynamically varying environments. The design of GOORB follows the basic concepts and the infrastructure employed in the OpenORB reflective architecture discussed above. In particular, each group type is represented as a template and each group instance is represented as a composite component. The constituents (see Fig. 2) of these composite components are as follows:

- *Group members* are composite components whose internal constituents are *application components* (e.g. databases access, audio and video playout etc.) combined with zero or more *system components* (e.g. stubs, skeletons, distributors, collators). Application components implement the basic functionality of the application whereas system components implement functionality associated with the GOORB group semantics.

– *Group bindings* are explicit representations of the topology and communication semantics of a given group type. It is not a default requirement that members of a group should use a specific communication pattern such as multicast. Although multicast is widely used as a support for groups, an application designer might equally well decide that members should communicate using several point-to-point communications, or any other composition of communication patterns. Thus, GOORB remains agnostic as to how members communicate among themselves while providing generic support for specifying and controlling this communication.
– *Management components* are system components that manage the group; e.g., components to control the membership of the group, or components to detect faults, or monitor the status of quality of service (QoS) of the group.

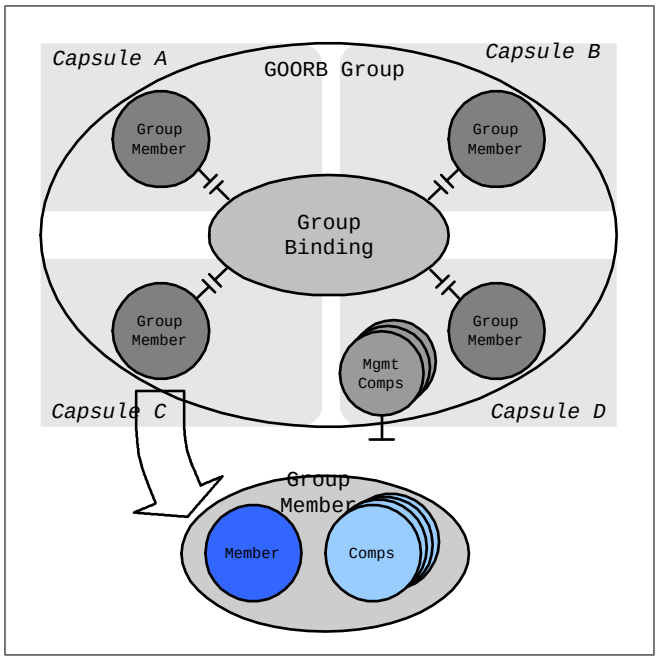

Fig. 2. A GOORB group

4 GOORB in Detail

4.1 Overview

The overall GOORB architecture, see Fig. 3, constitutes the following generic services: *i*) a *group binding service* which is responsible for providing functionality with which to configure and instantiate group bindings, *ii*) a *configurable*

group service which is responsible for the configuration, creation, destruction and control of group instances, and *iii*) a *group reconfiguration service* which supports the reconfiguration of group instances. All these services are accessed through a set of well-defined interfaces and rely on an extensible library of default components.

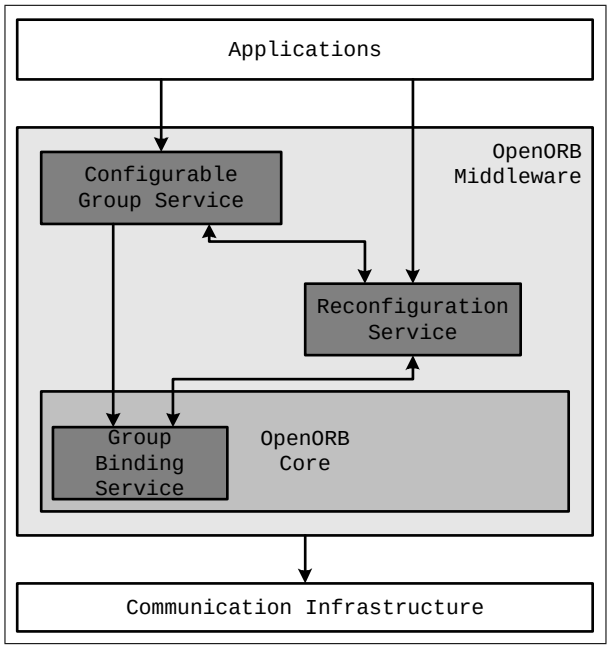

Fig. 3. The GOORB architecture

In the following subsections we discuss in detail the design and operation of the above three services.

4.2 The Group Binding Service

The group binding service is responsible for the creation and maintenance of group bindings. As mentioned, group bindings provide the core communications services in a GOORB group. However, it is important to notice that group bindings could also be used to realise interaction among components that are not part of a GOORB group. The difference between a GOORB group and a set of components connected via a group binding is the level of service provided. A GOORB group provides control over members of the group as default functionality while the group binding just enables the communication between several end-points. The group binding service can be considered a low-level enabler for groups while the GOORB group provides more comprehensive facilities.

Group bindings can be either primitive or composite. Primitive bindings implement or wrap basic communication functionality (e.g., IP multicast). These are directly supported by the infrastructure and their implementation is inaccessible to the programmer. Composite bindings then take the form of value-added encapsulations of one or more simpler bindings (e.g., IP multicast improved with components to guarantee reliability such as SRM [12] or RMT [14]). The basic building blocks of the group binding service architecture are the various factories. Factories for primitive group bindings are provided by the infrastructure and are parameterised with the references of the interfaces that will be connected by the binding. As well as setting up the protocol infrastructure necessary to support the group binding, they instantiate (in remote address spaces as required) necessary auxiliary system components such as stubs, controllers, etc. We have so far implemented the following primitive group binding factories:

- Multicast Stream Binding Factory. This creates primitive multicast stream bindings that can be used to connect several application interfaces, typically for the transmission of audio and video (e.g. to support an audio/video conferencing application). The application interfaces connected to these bindings can be either producers or consumers of audio/ video. An application component can be made both a producer and a consumer by implementing both interface types. The precise semantics of the binding can be flexibly configured. In particular, it is possible to create any of the following topologies: $1 - n$ (1 sender and n receivers), $n - m$ (n senders and m receivers), $n - 1$ (n senders and 1 receiver). Also, when several application interfaces are sending video, a receiving application can select which signals it wants to see.
- Multicast Operational Binding Factory. Primitive multicast operational bindings are used to bind multiple application interfaces in a request/ reply style of interaction. There are two roles defined: clients and servers. Clients make invocations on *stub* components, which forward the invocations to all currently bound server interfaces. By default, stubs block until the first reply arrives (cf. CORBA's deferred synchronous operation semantic [15]). However, this can be changed using a stub control interface so that the stub can instead be made to wait for any number of replies. All replies are then passed to a *collator* component. Different collators can implement alternative collation policies-i.e. alternative ways of combining replies so that a single consolidated reply can be presented to the client interface.
- Multicast Signal Binding Factory. Multicast signal bindings offers means for firing events in several capsules. Their structure is similar to that of multicast stream bindings; i.e., source components send events that are captured by sink components.

Each primitive group binding instance has a control interface that provides methods for dynamically adding and removing interfaces to the binding. As explained below, GOORB groups take advantage of this to realise changes in their membership.

Composite group bindings embody more complex communication semantics. For example, an operational primitive binding can be associated with encryption components to produce a secure binding. In addition, such a secure binding can be further improved with compression components to form yet another type of binding. Following the approach taken for general OpenORB composite components, configurations are described in terms of a set of components and bindings between them as explained in Sect. 2. These can be local bindings, which connect components in the same capsule, or distributed bindings, which connect components in different capsules. In the particular case of distributed groups, the configuration will typically include at least one distributed binding.

Unlike primitive group bindings, composite bindings are designed by application developers and are created by a single generic factory. A template, which embodies a description of the to-be-created composite binding, is sent to this factory which evaluates the internal details of the template and sends requests to other factories (component and binding factories) according to the configuration and location of the components to be instantiated to create the binding. The binding template is realised as an XML document. In more detail, configurations are specified by a graph that is formed by a list of components and a list of local bindings to be established between specified interfaces of these components. Because only local bindings are specifiable, nested distributed binding components must be used to bind any interfaces not located in a common capsule.

Finally, it is worth mentioning that the configurable group service framework includes an extensible selection of pre-defined generic protocol components that can be used in the specification of arbitrary composite bindings. For example, the stubs and collators mentioned above are included in this selection. Collators are, in fact, an instance of a more general class of components called *filters*. These support standard interfaces which allow them to be composable into stacks. Other examples of filters are *encryptors/ decryptors* and *compressors/ decompressors*. A further example is an *orderer* component which orders incoming messages according to some policy (e.g. for use in a totally ordered multicast binding).

4.3 The Configurable Group Service

In GOORB, group types are defined using XML-specified *group type templates* which specify the configuration and characteristics of to-be-created groups. These are similar in concept to the above-described Group Binding Service templates, but with additional complexity and functionality. Group type templates, which are passed to a generic group factory, include information about the types of potential members that can be part of the group, how these members communicate among themselves, the externally-visible services the group provides, any QoS parameters associated with the group etc. The template is an explicit representation of the group at creation time. In addition, though, a version of the template is retained during the lifetime of the group instance to facilitate reconfiguration (see Sect. 4.4). Essentially, the initial template describes the possible compositions that can be realised for the associated group type, while the

run-time version represents the actual configuration of a running group at any instant of time.

The XML Schema for group type templates is shown in Appendix A; an example of its use is given in Sect. 5. Essentially, the template embodies both *structure* and *constraints*: the group structure is represented by a graph defining components and the bindings between them; the constraints then define design-time and run-time limitations associated with the composition of the graph (e.g., the maximum or minimum number of elements of a certain type in the graph).

In more detail, the group template defines a group as consisting of a *group binding* configuration (as described above in Sect. 4.2), one or more *member types* (e.g., it may be useful to distinguish producer and consumer members in a media dissemination group), and zero or more *service types*. A member type is a template for the creation of future members of the group. Each consists of a *member interface type* and a *per-member configuration*. The member interface type is the type of interface that a prospective member of the associated member type will present to the group, while the per-member configuration is a graph of application and system components that is dynamically instantiated each time such a member joins the group. It is used to specify things like per-member stubs, skeletons, protocol stacks, collators, etc. In addition to the member interface type and per-member configuration, a member type can also include *attributes* that specify, e.g., the minimum and maximum number of instances of this type that may be in existence.

Service types are useful only in connection with "open groups" [16]. These are groups that provide a service that can be invoked by parties external to the group-e.g. in the case of a replicated service. As with member types it is possible to define multiple service types, and each service type similarly consists of an interface type, a per-service instance configuration, and attributes. "Closed groups", i.e. those with no service types, are also quite common: a typical example is a video conference.

The most important of the Configurable Group Service's run-time elements are the *generic group factory* and the *local factory*. The generic group factory interprets group type templates and calls on local factory instances, one of which runs on every participating node, to actually create the required components in the required location.

Each running group supports a default *group management* component that interacts with local factories to request the creation of new member and service instances. The generic group factory implicitly creates an instance of this default group management component type when the group is first created, and binds it into the group composite-unless another membership component (presumably with non-standard semantics) is defined in the group type template (it is also possible to change this later using the Reconfiguration Service). All group management components must implement at least the minimal *IGroup* interface that provides methods for components to join or leave a group. IGroup offers two methods for inserting members in a group and another two methods for removing them: *join()* inserts an already-instantiated actual member (i.e.

application components only; no system components) into the group while *createMember()* instantiates a set of application components in a specified capsule and then inserts them into the group. (This is similar to the *MembershipStyle* property defined in the CORBA-FT Specification [3] where the membership can be either "infrastructure-controlled" or "application-controlled".) *IGroup* also supports *leave()* and *destroyMember()* methods. In the case of *leave()* only the infrastructure is destroyed while in the case of *destroyMember()*, the member component is also destroyed. *IGroup* additionally provides a *getView()* method which returns the list of current members in the group.

4.4 The Reconfiguration Service

The final element of GOORB is its support for run-time reconfiguration of group instances for purposes of adaptation and evolution. This is based on a *compositional meta model* which is an extension of the basic OpenORB compositional meta-model mentioned in Sect. 2. In outline, an application wishing to perform a reconfiguration operation on a group obtains an interface to the group's compositional meta-model. It then manipulates this (causally-connected) meta-model-which takes the form of a topological graph plus operations to manipulate this graph-to effect corresponding changes in the actual underlying group. For example, nodes in the graph can be added, removed, and replaced to effect the corresponding addition, removal and replacement of actual underlying components (e.g. group members), as can arcs to effect the addition etc. of local bindings.

The implementation of the compositional meta-model mirrors the fact groups are inherently-distributed composite components-i.e. the compositional meta-model graph is itself decomposed and distributed. This helps with both scalability and fault tolerance. To further contribute to scalability, individual decomposed parts of the graph are dynamically instantiated on demand in an incremental way. The way in which remote access to meta-models is achieved is illustrated in Fig. 4. Basically, if the target component (or subcomponent) is non-local, requests to instantiate its compositional meta-model are transparently forwarded, by a local, per-capsule, *remote adaptor*, to the appropriate per-capsule *adaptation manager* in the capsule in which the target resides. Then, this adaptation manager requests a local factory (not shown in the figure) to create the local part of the target component's compositional meta-model and returns a reference to this. Finally, the remote adaptor can dynamically bind to the remote meta-model and invoke method calls to inspect and adapt the sub-graph (and hence the underlying component topology) just as if it were local.

Adaptation managers are responsible for ensuring that only permitted requests are processed and that only secure adaptations occur. To confirm the security of requested adaptations, adaptation managers call on per-group *adaptation data* components. These hold a set of adaptation constraints, expressed in XML and derived from the group template, for a particular group instance. For example, it is possible to forbid the deletion of parts of a per-member configuration such as a sub-composite that implements security checks. Because

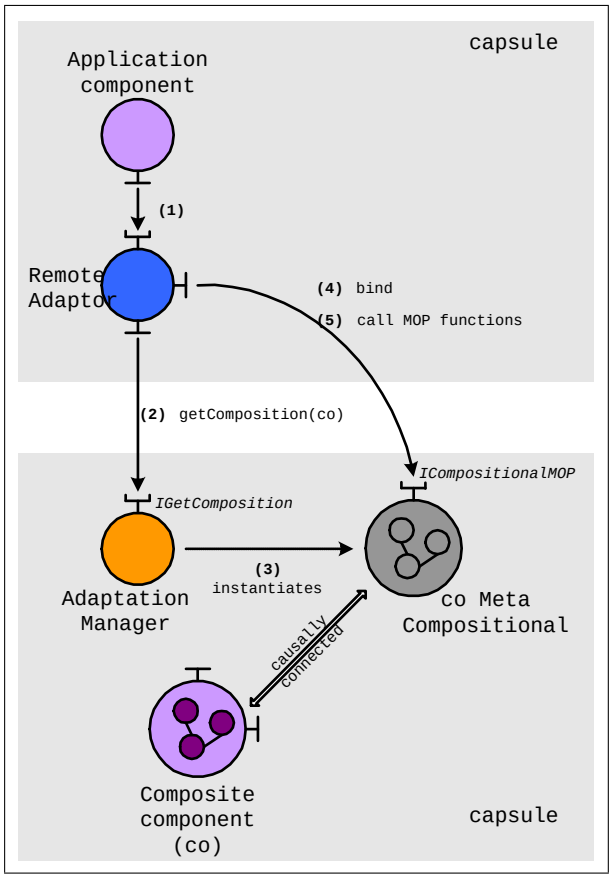

Fig. 4. The reconfiguration architecture

adaptation is a distributed operation, the constraints associated with a composite component are correspondingly distributed in per-capsule adaptation data components so that the failure of one host does not compromise the whole system. Additionally, local adaptations do not have to rely on information residing in another host.

Figure 4 actually presents an abstraction of the reconfiguration architecture. At a finer level of detail, two specific types of adaptation managers are employed: *group adaptation managers* (GAMs) and *member adaptation managers* (MAMs). These are respectively created when a group and members of a group are created. The GAM is responsible for receiving and delegating all adaptation requests to be performed on a group, while MAMs are responsible for performing adaptation on

a specific member of a GOORB group [1]. Essentially, GAMs and MAMs deal with two issues: transactional updates and replicated updates. Transactional updates are important to ensure that distributed group reconfiguration operations are performed either completely or not at all. Replicated updates, on the other hand, are useful to enable updates to be performed on all members or services in one action. For example, in one request one can add a compression component to all the receiving video members in a media dissemination group.

Finally, the reconfiguration architecture incorporates an *adaptation monitor*, which is a per-capsule composite component that embodies the OpenORB QoS management architecture [17]. This is capable of being configured to monitor some particular QoS metric (e.g. "the frequency of invocations on a replicated group service"), and then to select an appropriate adaptation strategy (e.g. "move from active to passive replication") when a particular trigger is detected (e.g. "more than n invocations per second are occurring"). Following the prototype presented in [17], monitors and strategy selectors are defined in terms of FC2 timed automata, while strategy activation consists of a script containing a sequence of calls on the group's compositional meta-model. Crucially, adaptation monitors can be dynamically instantiated in capsules so that it is not necessary that modes of adaptation in a group have been foreseen at design or deploy time (although these must not violate the adaptation constraints referred to above).

5 An Example of the Use of GOORB

In this section, we illustrate the use of GOORB by means of a simple dependability-related application scenario. The scenario involves a group of actively replicated servers (these are "calculators") in an open group with a single service type. The cardinality of the service type is restricted to a single instance, and there is a single member type with unconstrained cardinality to represent the replicants. The group template is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<group xmlns="http://site/GOORB/ggr"
       xmlns:gbs="http://site/GOORB/gbs"
       xmlns:ggb="http://site/GOORB/ggb"
       xmlns:ggm="http://site/GOORB/ggm"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://site/GOORB/ggr
       E:\Schema\GOORBGroup.xsd" groupName="ActiveReplication">
   <groupBinding groupBindingType="ReliableMulticastBinding.xml"/>
   <managementComponents/>
   <memberTypes>
      <member memberType="CalcReplica.xml" minCardinality="1" maxCardinality="unbounded"/>
   </memberTypes>
   <serviceTypes>
      <member memberType="CalcService.xml" minCardinality="1" maxCardinality="1"/>
   </serviceTypes>
</group>
```

[1] In the context of reconfiguration, service instances (as defined in Sect. 4.3) are considered members of the group. Therefore, each service instance has its own MAM as well as every member.

A reliable distributed binding (separately described in *ReliableMulticastBinding.xml*) is employed as the basis of communication within the group. *CalcReplica.xml* and *CalcService.xml* are templates that describe the configuration of the member and service types of the group respectively. *CalcService.xml*, which is shown below, contains a proxy component that receives the replies from the requests on the replicas and sends the first reply back to the client.

```
<?xml version="1.0" encoding="UTF-8"?>
<member xmlns="http://site /GOORB/ggm"
        xmlns:gbs="http://site/GOORB/gbs"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://site/GOORB/ggm E:\Schema\GroupMember.xsd">
   <gbs:configuration name="calcService">
      <gbs:graph>
         <gbs:component template="GetFirstCalcProxy.xml">proxy</gbs:component>
      </gbs:graph>
      <gbs:externalInterface>
         <gbs:bindable>
            <gbs:componentName>proxy</gbs:componentName>
            <gbs:interfaceName>ICalc__</gbs:interfaceName>
         </gbs:bindable>
         <gbs:bindable>
            <gbs:componentName>proxy</gbs:componentName>
            <gbs:interfaceName role="service">ICalc</gbs:interfaceName>
         </gbs:bindable>
      </gbs:externalInterface>
   </gbs:configuration>
   <externalBinding>
      <gbs:localBinding>
         <gbs:bindable>
            <gbs:componentName>proxy</gbs:componentName>
            <gbs:interfaceName role="server">ICalc__</gbs:interfaceName>
         </gbs:bindable>
         <gbs:bindable>
            <gbs:componentName>mob</gbs:componentName>
            <gbs:interfaceName>iface</gbs:interfaceName>
         </gbs:bindable>
      </gbs:localBinding>
   </externalBinding>
</member>
```

As explained in Sect. 4, the group template is passed as a parameter to the generic group factory's *create()* method. After the group has been created, it can be populated with member and service instances, as explained in Sect. 4.3, by calling operations the *join()* or *createMember()* operations on the (default) group management component's *IGroup* interface.

Having established and populated the group, it is possible to adapt it by employing the Reconfiguration Service. As a somewhat contrived example (chosen for simplicity of exposition), it may be desired to replace the initial proxy component mentioned above with one that returns the result of a voting process between the replicas. To achieve this, the client application (or a strategy activator if reconfiguration is being initiated automatically by an embedded QoS manager) needs to access the group's GAM's *IGAdapt* interface in order to invoke its *adaptType()* method (this enables an adaptation to be atomically performed on every member of the specified type). *AdaptType()* takes as a parameter the name of an adaptation script which, in our case, contains the following single call:

```
comp_meta_model.replaceComponent("proxy", {"name":"vproxy",
                                           "template":"VotingProxy.xml",
                                           "module":"Comps.VotingProxy"});
```

As a result of executing this script, every member of the group has its original proxy component replaced by a new "vproxy" component. The compositional meta-model's *replaceComponent()* method substitutes a component in the group composite for another component of the same type, i.e., one with the same set of provided and required interfaces. The first parameter is the component to be removed and the second is either the new component (if this has already been instantiated), or a template for a component that needs to be created (as shown in the example). Both the *IGAdapt* interface and the more general facilities of the compositional meta-model are discussed in more detail in [18].

6 Related Work

Adaptive fault tolerance has been described by Kalbarczyk [19] as the ability for a system to adapt dynamically to changes in the fault tolerance requirements of an application. Some issues of adaptive fault tolerance have been addressed by projects such as *Chameleon* [19], *AFTM* [20] and *Proteus* [21] (part of the AQuA system). Chameleon is an infrastructure that supports multiple fault tolerance strategies. These are realised by objects called ARMORs which can be used (and reused) to build different fault tolerance strategies. Furthermore, ARMORs can be added/removed or changed at run-time, thus providing dynamic reconfiguration. Such reconfiguration, however, is aimed at modifying the fault tolerance level only-it is less general than the capabilities offered by GOORB. In the AFTM (Adaptive Fault Tolerant Middleware) [20], several fault-tolerant execution modes are available, from which the most suitable mode can be selected according to application's requirements. However, only a small selection of execution modes is available in the AFTM environment and there are no guidelines on how to develop new modes-in other words, unlike GOORB, the system in not easily extensible. The Adaptive Quality of Service for Availability (AQuA) [21] architecture provides a flexible infrastructure for developing dependable systems. It relies on a group of technologies to provide applications with their required availability. First, it uses the Quality Objects (QuO) framework for specifying dynamic quality of service (QoS) requirements. Second, it uses the Proteus framework to handle fault tolerance through replication. Finally, at the lowest level of the platform, AQuA uses the Maestro/Ensemble [22] group communication system to provide reliable multicast to a process group. Although AQuA is highly flexible in terms of the configuration, it lacks support for run-time reconfiguration.

A number of researchers have attempted to build group services in terms of components (in a loose interpretation of the term). Notable examples are the Ensemble toolkit from Cornell University [22], work at Michigan [23] and the "building block" based reliable multicast protocol proposed by the IETF [14]. However, these efforts are primarily targeted at low levels aspects of group provision (i.e. communications services) and their component models are far less

general than ours. A more general approach is proposed by projects such as Coyote [24], Cactus [25] and Apia [26], however, they also address low-level protocol stack issues.

At a higher system level, the Object Management Group (OMG) has defined a multi-party event service [4] for CORBA and has recently added fault tolerance by replication of objects to its specification [3]. However, as mentioned in the introduction, these efforts are limited in scope and fail to address the needs of the full diversity of group applications. This can also be said of a number of other group-oriented efforts in the CORBA world. For example Electra [27], Eternal [28], OGS [29], OFS [30] and NewTop Object Group Service [31] are all targeted solely at fault tolerant application scenarios and cannot easily be employed in the construction of other types of group application. Furthermore, they tend to provide flexibility though the setting of per-defined properties, which, naturally enough, represent only those degrees of freedom envisaged by the designer of the system. There is no support for the definition of entirely new group services that meet as yet unforeseen needs.

Groups have been developed for middleware platforms other than CORBA such as JGroup [32] for the Java RMI (Remote Method Invocation), the work described in [33] in the context of the Regis platform, and ARMADA [34] which is another middleware system designed specifically for fault tolerance. Similar comments to the above can be applied to all these systems. ARMADA explicitly address the need for adaptivity, but not though a component/ reflection-based approach.

Work at Cornell on the Quintet system [35] uses COM components to build reliable enterprise services. This work recognises an increased need for flexibility (e.g. rather than prescribe transparency, they allow groups to be explicitly configured), but it is not as radical as our reflection-based approach. In addition, the work is not targeted at the full range of group applications.

Our component model is influenced by models such as COM+ [2], Enterprise JavaBeans [36] and CORBA Components [15], all of which support similar container-based models for the construction of distributed applications. We add a number of novel aspects to such models including the support of multiple interaction types, and the notion of local/ distributed bindings. We also add sophisticated reflection capabilities in the form of our multiple meta-model approach.

7 Conclusions

This paper has described a flexible architecture for building groups with a wide diversity of requirements. In particular, the architecture aims to address three main areas: i) providing group support in a generic middleware environment; ii) the need for flexibility in creating groups, and iii) the further need for flexibility during the group operation. Our experience to date with the architecture has been very favourable: we have been able to build, with minimal effort, a wide range of group services using only a relatively limited set of base compo-

nents. In this paper, the focus has been on dependability scenarios; however, it is important to emphasise that GOORB has been successfully applied in other group-related areas. For example, its application in a multimedia group-based scenarios is discussed in [37].

As mentioned, XML templates serve as the basis for both group creation and for restricting reconfiguration at run-time (where required) by means of constraints inserted in the template. However, this heavy reliance on XML templates raises a possible drawback of our design: the potential complexity of templates; particularly for large and complex groups. To alleviate this we are currently investigating the provision of 'front-end factories' that are specially designed for different application areas (e.g. replicated services, or multimedia groups). Building on the current generic factory/ template architecture, these will provide a custom interface and map to the standard XML templates accepted by the generic group factory. Another important extension that we are investigating is the use of a graphical front-end as a means of composing groups from components in repositories. We also envisage that this can be used at run-time to initiate reconfiguration operations.

In a future implementation phase we will migrate our implementation to the OpenORB v2 environment [7]. This performs significantly better than the Python based prototype that is discussed in this paper (it is based on a binary level inter-component communication scheme, called OpenCOM [38], which is superset of a subset of Microsoft's COM). The use of OpenORB v2 will permit more realistic experimentation with media dissemination groups and will also allow us to address questions regarding the performance implications of our approach.

A final area of future work is to investigate in more detail the full power of the reflective capability of groups. apart from its use in reconfiguration as discussed in this paper, the use of reflection opens up many further possibilities; e.g., passivating and activating groups, inserting quality of service monitors into group configurations, or add interceptors to perform logging of messages. There are many unsolved problems in this area, not the least of which is the difficulty of maintaining the integrity of configurations when they are adapted at run-time [39]. However, it seems a highly desirable goal to offer such facilities.

References

[1] Object Management Group: CORBA Object Request Broker Architecture and Specification - Revision 2.3 (1999)

[2] Microsoft Corporation: COM Home Page. Available at <http://www.microsoft.com/com/> (1999)

[3] Object Management Group: Fault Tolerant CORBA Specification, V1.0. Available at <http://www.omg.org> (2000) OMG document: ptc/2000-04-04.

[4] Object Management Group: Event service, v1.0. Formal/97-12-11 (1997)

[5] Blair, G., Coulson, G., Robin, P., Papathomas, M.: An Architecture for Next Generation Middleware. In: Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer-Verlag (1998) 191–206

[6] Costa, F.M., Duran, H.A., Parlavantzas, N., Saikoski, K.B., Blair, G., Coulson, G.: The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications. In Cazzola, W., Stroud, R.J., Tisato, F., eds.: Reflection and Software Engineering. Lecture Notes in Computer Science 1826. Springer-Verlag, Heidelberg, Germany (2000) 79–99
[7] Coulson, G., Blair, G., Clark, M., Parlavantzas, N.: The Design of a Highly Configurable and Reconfigurable Middleware Platform. ACM Distributed Computing Journal **15** (2002) 109–126
[8] Szyperski, C.: Component Software - Beyond Object-Oriented Programming. Addison-Wesley (1998)
[9] Coulson, G., Blair, G., Davies, N., Robin, P., Fitzpatrick, T.: Supporting Mobile Multimedia Applications through Adaptive Middleware. IEEE Journal on Selected Areas in Communications **17** (1999) 1651–1659
[10] Maes, P.: Concepts and Experiments in Computational Reflection. In: Proceedings of OOPSLA'87. Volume 22 of ACM SIGPLAN Notices., ACM Press (1987) 147–155
[11] ISO/IEC: Open Distributed Processing Reference Model, Part 1: Overview. ITU-T Rec. X.901 — ISO/IEC 10746-1, ISO/IEC (1995)
[12] Floyd, S., Jacobson, V., Liu, C., McCanne, S., Zhang, L.: A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. IEEE/ACM Transactions on Networking **5** (1997) 784–803
[13] World Wide Web Consortium: Extensible Markup Language (XML) 1.0. W3C Recommendation (1998)
[14] Whetten, B., Vicisano, L., Kermode, R., Handley, M., Floyd, S., Luby, M.: Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer. INTERNET-DRAFT - RMT Working Group, Internet Engineering Task Force (2000) draft-ietf-rmt-buildingblocks-02.txt.
[15] Object Management Group: CORBA Components Final Submission. OMG Document orbos/99-02-05 (1999)
[16] Olsen, M., Oskiewicz, E., Warne, J.: A Model for Interface Groups. In: Proceedings of IEEE 10th Symp. on Reliable Distributed Systems. (1991)
[17] Blair, G.S., Andersen, A., Blair, L., Coulson, G.: The role of reflection in supporting dynamic QoS management functions. In: Seventh International Workshop on Quality of Service (IWQoS '99). Number MPG-99-03 in Distributed Multimedia Research Group Report, London, UK, IEEE/IFIP, Lancaster University (1999)
[18] Saikoski, K., Coulson, G.: Experiences with OpenORB's Compositional Meta-Model and Groups of Components . In: The Workshop on Experience with Reflective Systems, Kyoto, Japan (2001)
[19] Kalbarczyk, Z.T., Bagchi, S., Whisnant, K., Iyer, R.K.: Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. IEEE Trans. on Parallel and Distributed Systems **10** (1999) Special Issue on Dependable Real Time Systems.
[20] Shokri, E., Hecht, H., Crane, P., Dussalt, J., Kim, K.: An Approach for Adaptive Fault Tolerance in Object-Oriented Open Distributed Systems. In: Proceedings of the Third International Workshop on Object-oriented Real-Time Dependable Systems (WORDS'97), Newport Beach, California (1997)
[21] Sabnis, C., Cukier, M., Ren, J., Rubel, P., Sanders, W.H., Bakken, D.E., Karr, D.A.: Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA. In: Proceedings of the 7th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-7), San Jose, CA, USA (1999) 137–156

[22] van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., Karr, D.: Building Adaptive Systems Using Ensemble. Technical Report TR97-1619, Cornell University (1997)

[23] Litiu, R., Prakash, A.: Adaptive Group Communication Services for Groupware Systems. In: Proceedings of the Second International Enterprise Distributed Object Computing Workshop (EDOC'98), San Diego, CA (1997)

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

[25] Hiltunen, M., Schlichting, R.: The Cactus Approach to Building Configurable Middleware Services. In: Proceedings of the SRDS Dependable System Middleware and Group Communication Workshop (DSMGC), Nürnberg, Germany (2000)

[26] Miranda, H., Pinto, A., Rodrigues, L.: Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels. In: Proceedings of the 21st International Conference on Distributed Computing Systems, Phoenix, Arizona, IEEE (2001) 707–710

[27] Maffeis, S.: Adding Group Communication Fault-Tolerance to CORBA. In: Proceedings of USENIX Conference on Object-Oriented Technologies, Monterey, CA (1995)

[28] Narasimhan, P., Moser, E., Melliar-Smith, P.M.: Replica consistency of CORBA objects in partitionable distributed systems. Distributed Systems Engineering Journal **4** (1997) 139–150

[29] Felber, P.: The CORBA Object Group Service: A Service Approach to Object Groups in CORBA. PhD thesis, Department D'Informatique – École Polytechnique Fédérale de Lausanne (1997)

[30] Sheu, G.W., Chang, Y.S., Liang, D., Yuan, S.M., Lo, W.: A Fault-Folerant Object Service on CORBA. In: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97), Baltimore, MD (1997)

[31] Morgan, G., Shrivastava, S., Ezhilchelvan, P., Little, M.: Design and Implementation of a CORBA Fault-tolerant Object Group Service. In: Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinki, Finland (1999)

[32] Montresor, A.: The Jgroup Reliable Distributed Object Model. In: Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinki, Finland (1999)

[33] Karamanolis, C., Magee, J.: A Replication Protocol to Support Dynamically Configurable Groups of Servers. In Press, I.C.S., ed.: Proceedings of the Third International Conference on Configurable Distributed Systems (ICCDS'96), Annapolis MD (1996)

[34] Abdelzaher, T.F., Dawson, S., Feng, W.C., Jahanian, F., Johnson, S., Mehra, A., Mitton, T., Shaikh, A., Shin, K.G., Wang, Z., Zou, H.: ARMADA Middleware and Communication Services. Real-Time Systems **16** (1999) 127–153

[35] Vogels, W., Dumitriu, D., Pantiz, M., Chipawolski, K., Pettis, J.: Quintet, Tools for Reliable Enterprise Computing. In: Proceedings of the 2nd International Enterprise Distributed Object Computing Workshop (EDOC '98), San Diego, CA (1998)

[36] Sun Microsystems: Enterprise JavaBeans Specification Version 1.1. Available at <http://java.sun.com/products/ejb/index.html> (2000)

[37] Saikoski, K.B., Coulson, G., Blair, G.: Configurable and Reconfigurable Group Services in a Component Based Middleware Environment. In: Proceedings of the SRDS Dependable System Middleware and Group Communication Workshop (DSMGC), Nürnberg, Germany (2000)
[38] Clark, M., Blair, G., Coulson, G., Parlavantzas, N.: An Efficient Component Model for the Construction of Adaptive Middleware. In: Proceedings of the IFIP Middleware Conference 2001. Volume 2218., Heidelberg, Germany, Lecture Notes in Computer Science (2001)
[39] Blair, G., Coulson, G., Andersen, A., Blair, L., Clark, M., Costa, F., Limon, H.D., Parlavantzas, N., Saikoski, K.B.: A Principled Approach to Supporting Adaptation in Distributed Mobile Environment. In: 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000), Limerick, Ireland (2000)

APPENDIX A: XML Schema for GOORB Groups

Below is the standard XML Schema for describing GOORB groups. As can be seen, a group is a composition of group members and services, a group binding and management components. Some of these elements are described in other XML schemas as follows: OpenORB's basic elements such as composite components, interfaces and local bindings are described in *GOORBBasics.xsd* (*gbs* namespace); group bindings are described in *GroupBinding.xsd* (*ggb* namespace); and group members and services are described in *GroupMember.xsd* (*ggm* namespace). Note that, for ease of reuse, separate templates can be used to describe some of the elements. For example, a group member can be included either as a separate template or directly described in the group template itself. The same option can be used for service types and group bindings.

Another important issue is related to the constraints associated to groups. For example, it is possible to restrict the cardinality of the group or the cardinality of a specific member type. Another set of constraints, which are not shown in the schema below, is related to adaptation. For example, it is possible to define if adaptation can be realised in a particular group or what level of adaptation can be realised. Each set of rules (constraints) is associated to a number of management components. New set of rules requires new versions of these components.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://site/GOORB/ggr"
xmlns:ggm="http://site/GOORB/ggm" xmlns:ggr="http://site/GOORB/ggr"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:gbs="http://site/GOORB/gbs"
xmlns:ggb="http://site/GOORB/ggb"
elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xs:import namespace="http://site/GOORB/gbs" schemaLocation="GOORBBasics.xsd"/>
   <xs:import namespace="http://site/GOORB/ggb" schemaLocation="GroupBinding.xsd"/>
   <xs:import namespace="http://site/GOORB/ggm" schemaLocation="GroupMember.xsd"/>
   <xs:element name="group">
      <xs:annotation>
         <xs:documentation>A group is a composition of group members and services, a
                           group binding and management components
         </xs:documentation>
      </xs:annotation>
```

```
   <xs:complexType>
      <xs:sequence>
         <xs:choice>
            <xs:element ref="ggb:groupBinding"/>
            <xs:element ref="ggr:groupBinding"/>
         </xs:choice>
         <xs:element ref="ggr:managementComponents"/>
         <xs:element ref="ggr:memberTypes"/>
         <xs:element ref="ggr:serviceTypes"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="memberTypes">
   <xs:annotation>
      <xs:documentation>A member of the group</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="ggr:participantType" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="serviceTypes">
   <xs:annotation>
      <xs:documentation>A service offered by the group</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="ggr:participantType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="participantType">
   <xs:annotation>
      <xs:documentation>A participant in the group (service or member)
      </xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:choice>
         <xs:element ref="ggr:member"/>
         <xs:element ref="ggm:member"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="type" type="xs:anyURI" use="required"/>
      <xs:attribute name="minCardinality" type="xs:integer" use="required"/>
      <xs:attribute name="maxCardinality" type="xs:integer" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="groupBinding">
   <xs:annotation>
      <xs:documentation>A reference to an external groupBinding xml file
      </xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:attribute name="groupBindingType" type="xs:anyURI" use="required"/>
   </xs:complexType>
</xs:element>
<xs:element name="managementComponents">
   <xs:annotation>
      <xs:documentation>A list of management components</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="gbs:component" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="member">
```

```
      <xs:annotation>
         <xs:documentation>A reference to an external member</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:attribute name="memberType" type="xs:anyURI" use="required"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```