

# Configurable and Reconfigurable Group Services in a Component Based Middleware Environment

Katia B. Saikoski, Geoff Coulson and Gordon Blair  
Distributed Multimedia Research Group,  
Department of Computing, Lancaster University,  
Lancaster, LA1 4YR, U.K.  
Phone: +44 (0)1524 593801 / Fax: +44(0) 1524 593608  
{saikoski,geoff,gordon}@comp.lancs.ac.uk  
<http://www.comp.lancs.ac.uk>

## Abstract

The past few years have seen a significant increase in the importance of *group based* distributed applications such as media dissemination, computer supported collaborative work or fault tolerance through replication. However, most distributed object based middleware platforms, which are increasingly being used as an implementation environment for such applications, fail to provide suitable support for group applications in their full generality. In this paper we describe a *component based* approach to the provision of group services in a middleware environment in which application tailored group services can be built by defining particular configurations of components or by incrementally modifying existing configurations. In addition, our approach uses reflective capabilities of the middleware platform to support the run-time reconfiguration of existing and running group applications.

**Keywords:** group service, component-based systems, middleware, configurable groups; reflection.

## 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 endpoints, the class of group applications is characterised by great diversity. Included in the class, for example, are such diverse applications as dissemination of audio and

video, distribution of events, computer supported cooperative work, and fault-tolerant or highly available applications based on replication of servers.

Unfortunately, present day distributed object based middleware platforms (e.g. CORBA [24] or DCOM [18])<sup>1</sup>, which are increasingly being used as an implementation environment for such applications, fail to provide suitable support for group applications in their full generality. If they provide any support at all it is typically targeted at a limited subset of group applications. Examples from the CORBA world are the (completely separate) CORBA event service [22], and the recently developed fault tolerance service [25]. This lack of integration, although understandable for reasons of pragmatism, is problematic in a number of ways. For example, 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 aim of our work 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 application 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 multi-

---

<sup>1</sup>For the purposes of this paper, the term middleware refers to a distributed software layer, or “platform” which abstracts over the complexity and heterogeneity of the underlying distributed environment with its various network technologies, machine architectures, operating systems and programming languages.

cast services etc.

Our approach is to satisfy all such requirements in a middleware environment known as *OpenORB* which, in addition to supporting (group) applications, also supports the development of application tailored group services. This is primarily achieved through the use of *component technology* [31] which allows programmers to build application tailored group services in terms of an extensible library of basic building blocks (components). Components are used not only at the application level; the middleware itself is built from components. Interestingly, this allows groups to be used recursively to support communication among the various middleware components themselves. For 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 group services. As an example, consider a scenario involving 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 and the performance/ reliability needs change over time. Another example could involve adding or removing media filters such as video compression components to support media dissemination in multimedia conferences involving nodes with heterogeneous network connectivity and processing capability [7]. To support such requirements, we apply the notion of *reflection*<sup>2</sup> [16]. In particular, we maintain run-time *component graphs* which allow component configurations to be adapted in a “direct manipulation” style (see section 2).

This paper is structured as follows. Firstly, section 2 describes our basic middleware architecture and its reflective, component based, computational model. This is presented in some detail as the group architecture heavily depends on it. Subsequently we discuss, in section 3, our approach to the construction of group services in the context of the component model. Section 4 then presents some implementation details. Finally, section 5 deals with related work and we offer concluding remarks in section 6.

---

<sup>2</sup>Reflection is the capability of a system to reason about and act upon itself. A reflective system contains a representation of its own behaviour, amenable to examination and change, and which is causally connected to the behaviour it describes. “Causally connected” means that changes made to the system’s self-representation are immediately reflected in its actual state and behavior, and vice-versa.

## 2 The OpenORB Component Based Middleware Architecture

As mentioned above, OpenORB [4, 6] is built according to a component-based architecture. At load-time, components are selected and appropriately composed to create a specific instance of the middleware. For example, components encapsulating threads, buffer management, the IIOP protocol and the Portable Object Adapter may be selected, among others, for placement within a CORBA capsule (i.e. address space). In addition, components can be loaded into capsules at run-time and, because they interact via a binary level standard, there are no restrictions on the language in which they are written. As explained below, reflection can be used to facilitate the change and reconfiguration of the set of components in a capsule at run-time, and thus dynamically adapt the middleware functionality.

In our architecture, 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, any number of instances of these types can be created on demand at run-time. Multiple interface types provide 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 instances of these multiple types are then useful for giving multiple clients their own private “view” of the component.

In addition to standard operational interactions (i.e. method calls), we support *signal* and *stream* interaction types as defined in ISO RM-ODP [12]. Typically, signal interactions are used for primitive events and stream interactions for audio or video etc. RM-ODP claims 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*: *provided* or *required*. *Provided* interfaces represent the services offered to the component’s environment, while *required* interfaces represent the services the component requires from its environment (in terms of the *provided* interfaces of other components). This explicit statement of requirement 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 either explicitly or implicitly *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.). Crucially, bindings between interfaces are themselves components; see figure 1. There are two categories of binding component: *local bindings* and *distributed bindings*. The former, which are simple and primitive in nature, are 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 themselves are *composite* and *distributed* components which may span capsule or machine boundaries. Internally, these bindings are composed of *sub-components* (bound by means of local bindings) that typically represent aspects of the communications system. Examples are primitive transport 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 “video binding” may be created by encapsulating a configuration consisting of a “primitive” multicast IP binding augmented with H.263 filter components.

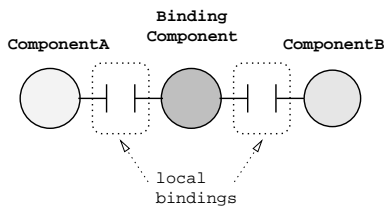


Figure 1: Binding components represent the interaction between components.

It should be noted that distributed bindings are not unique in being distributed and composite; any 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 *factories*). Factories for composite components are typically implemented as compositions of factories for the composite component’s various constituents. In addition, each capsule has a generic factory which allows capsules to be instantiated in that capsule. If the code for a given component type is not available in a given capsule, it can be downloaded on demand from elsewhere. Factories are parameterised by means of *templates* which define the required configuration and characteristics of to-be-created components. Templates are specified in the Extensible

Markup Language (XML) [37] and are further discussed in section 3.1 below.

In terms of reflection, every OpenORB component has an associated *meta-space*, which 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 *encapsulation meta-model* allows access to the component’s interfaces and their methods and bindings, and the *resources meta-model* gives access to the resources (e.g. threads or memory) used by the (non distributed) component. The meta-model of most relevance to this paper is the *compositional meta-model*. This, as mentioned above, takes the form of a *component graph* data structure which serves as a causally connected self-representation of a composite component’s internal structure. This means that manipulations of the component graph’s topology result in corresponding changes in the component’s composition. 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 detail in [4, 6].

## 3 The Construction of Group Services

### 3.1 Architecture

The construction of group services is heavily predicated on the availability of composite and distributed components. Given this support, it is natural to view groups as composite components and group members as sub-components of the component representing the group.

Following ANSA [26], we distinguish *closed* and *open* groups. With closed groups, no interaction is required with the environment outside the group. An example of such a group could be a conference in which the members were registered participants. Open groups, on the other hand, have externally visible interfaces which allow external components to interact with the group. An example could be a replicated service made up of a group of server components which is accessible through a single service interface. Open groups can support either required or provided external interfaces (or both).

Our approach to providing both open and closed group services in the OpenORB environment is to offer:

- i) one or more specialised *group factories* whose template

schema appropriately supports the notions of open and closed groups, and

- ii) a library of *base components* (such as: group management, basic communications, ordering protocols, collation policies etc. [28]) from which groups can be composed by a group factory according to given templates.

Our prototype group factory provides the following interface (see listing 1); the template schema accepted by the factory, defined as an XML Document Type Definition (DTD), is presented in listing 2.

Listing 1: IDL for the GroupFactory interface

```
// CORBA IDL
interface GroupFactory {
    typedef string ComponentID;
    typedef string Template;

    ComponentID createGroup(in Template template);
};
```

Listing 2: DTD for templates

```
<?xml version="1.0" ?>
<!-- DTD templates for groups -->
<!ELEMENT GroupFac:template (configuration,
                             member_types,
                             service_types)>

<!ELEMENT member_types (participant_type+)>
<!ELEMENT service_types (participant_type*)>

<!ELEMENT participant_type (interface_type,
                             configuration?)>

<!ATTLIST participant_type
  name CDATA #REQUIRED
  min_cardinality CDATA #IMPLIED
  max_cardinality CDATA #IMPLIED>

<!ELEMENT configuration (component*,
                          local_binding*)>
<!ELEMENT local_binding (bindable,
                          bindable)>
<!ELEMENT bindable (component,
                    interface_type)>

<!ELEMENT interface_type (#PCDATA)>
<!ELEMENT component (#PCDATA)>
<!ATTLIST component
  template CDATA #IMPLIED>
```

The group factory’s template schema defines a group as consisting of a *global configuration*, one or more *member\_types* and zero or more *service\_types*. The global configuration is a configuration of components that is instantiated when the group is created; an example could be one or more distributed binding components of various types. It is specified as 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, distributed binding components must be used to bind any interfaces not located in a common capsule. Note that all components appearing in a template specification (optionally) have their own template attached as an attribute.

A *member\_type* is essentially a template for future members of this group. It is possible to specify any number of *member types* (e.g. it is useful to distinguish producers and consumers in a media dissemination group). Each *member\_type* consists of an *interface\_type* and a per-member *configuration*. The latter is used to specify things like per-member protocol stacks, filters and collators etc. (see section 4). There are also attributes associated with each *member\_type* that permit the specification of a name for the *member\_type* and the minimum and maximum number of instances of this type that may be in existence. The *interface\_type* is the type of interface that a prospective member of the associated *member\_type* must provide to group management on joining the group (group management is discussed below). The *configuration* is a graph of components (similar in structure to the global configuration) that is dynamically instantiated each time a member of the associated *member\_type* joins the group. The interface of the prospective member is bound to an interface of this per-member configuration (as defined in the template) and, further, one or more interfaces of the per-member configuration are bound to interfaces of the global configuration, in order to bind the new member into the group topology.

*Service\_types* are used in connection with open groups; a “service” represents an externally visible view onto the group. As with *member\_types* it is possible to have multiple *service\_types*, each of which consists of an *interface\_type* and an associated per-service-instance *configuration* that is bound to the global configuration when a new service instance is created.

Each group created by the group factory supports the following default group management interface (see listing 3). This interface is provided by a default group management component that acts as a factory for new instances of the member and service configurations specified in the template.

Listing 3: IDL Interface for the GrMgmt

```
// CORBA IDL
interface GrMgmt {
    struct IRef {...};
    typedef sequence<IRef> IRefList;

    void join(in member_type memtype,
             in IRef iref);
    void leave(in IRef member)
```

```

IRefList getMembers();

IRef newService(in interface_type servtype);
void deleteService(in Iref service);
IRefList getServices();
};

```

The *join()*, *leave()* and *getMembers()* operations provide basic membership management. Each time *join()* is called, the given interface is bound to a newly instantiated per-*member\_type* configuration which, in turn, is bound to the global configuration as specified in the group's template. The *leave()* operation reverses this procedure and the *getMembers()* operation returns a current list of members. Similarly, the *newService()*, *deleteService()* and *getServices()* operations provide basic service management; *newService()* returns an interface representing a new instance of the given service type, *deleteService()* destroys an existing service instance, and *getServices()* returns a list of all currently active service interface instances.

Note that the GrMgmt interface is intended to support only basic group management functions. If a given group service requires additional management functionality, then an appropriate interface would be defined and configured as a service type supported by the group. The GrMgmt service would then serve only as a bootstrap mechanism through which the enhanced group management service type could be obtained.

Note also that the full power of the OpenORB reflective capabilities are available on groups. For example, the compositional meta-model can be used to inspect a group's topology and the encapsulation meta-model to obtain information on the types and capabilities of all the interfaces and local bindings involved. Furthermore, the group can be adapted and reconfigured as desired through the various reflective meta-models.

### 3.2 Examples

In this section, we present some examples which illustrate the generality of the group architecture by demonstrating its applicability to a diverse range of group scenarios.

The first example (see figure 2 and listing 4) employs a very simple group which is primarily intended to illustrate the use of the group factory's template schema. It is an open group with a single service type, the cardinality of which is restricted to a single instance, and a single member type with unconstrained cardinality. A single distributed binding, which has a control interface, *ctrl*, is employed as the global configuration.

Listing 4: Example of a template in XML

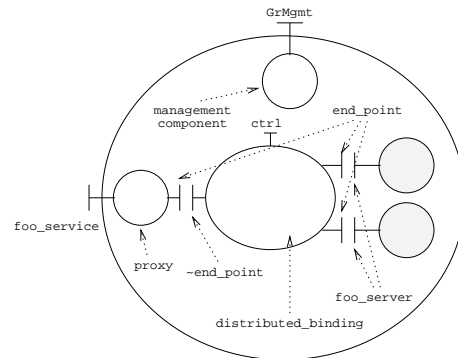


Figure 2: Example of a group.

```

<?xml version = "1.0" standalone = "no" ?>
<!DOCTYPE GroupFac:template SYSTEM "template.dtd">
<GroupFac:template>
  <configuration>
    <component>
      distributed_binding
    </component>
  </configuration>

  <member-types>
    <participant_type name = "foo_server">
      <interface_type>foo_server
      </interface_type>
      <configuration>
        <local_binding>
          <bindable>
            <component>this</component>
            <interface_type>foo_server
            </interface_type>
          </bindable>
          <bindable>
            <component>distributed_binding
            </component>
            <interface_type>end_point
            </interface_type>
          </bindable>
        </local_binding>
      </configuration>
    </participant_type>
  </member-types>

  <service_types>
    <participant_type name = "foo_service"
      min_cardinality = '0'
      max_cardinality = '1'>
      <interface_type>foo_service
      </interface_type>
      <configuration>
        <component>proxy</component>
        <local_binding>
          <bindable>
            <component>proxy</component>
            <interface_type>-end_point
            </interface_type>
          </bindable>
          <bindable>
            <component>distributed_binding
            </component>
            <interface_type>end_point

```

```

        </interface_type>
    </bindable>
</local_binding>
</configuration>
</participant_type>
</service_types>
</GroupFac:template>

```

The second example, which employs a closed group, illustrates an audio/ video dissemination scenario (see figure 3). In this case, we omit the template due to space constraints. The scenario employs three distinct member types, each with their own per-member-type configuration (AF and VF stand for audio and video filter respectively). The global configuration employs separate distributed bindings for audio and video.

Finally, our third example illustrates two fault tolerance scenarios; one (figure 4(a)) involving active server replication and the other (figure 4(b)) passive replication. An open group is employed in both cases. In the active replication scenario, a single service type (with single cardinality) is associated with a per-service-type configuration involving a proxy, stub and filter (F). The service type exports the same interface type as the group members (i.e. the replicated servers) and provides a transparent view of the replicated service. Note that the proxy could usefully contain a collation function (see section 4). The single member type (of unconstrained cardinality) is associated with a slightly simpler per-member-type configuration consisting only of a stub and a filter.

The passive replication scenario (figure 4(b)) is constructed from a similar set of components but combines them differently. This scenario also differs in that it involves a *nested group*. Nesting, or encapsulation, is a powerful mechanism for abstraction and reuse of group services which is realised by means of recursive applications of the group factory. In this scenario, the outer group supports a single service type, the configuration of which employs a component (labelled “Primary”) which is of the same type as the single member type (labelled “Backup”). The primary and backups are bound to instances of the single service type of the inner, nested, group. The inner group is used to support a state transfer protocol running between the primary and the multiple backups.

## 4 Implementation

Our current implementation environment is a prototype of OpenORB written in the Python interpreted language. This prototype [2] implements the component architecture described in section 2 with the limitation that only compo-

nents written in Python are supported. A number of middleware related components are deployed in the prototype which conspire to approximate a CORBA programming environment.

The environment also provides the necessary bootstrapping machinery to underpin reliable groups. Following the CORBA fault tolerance specification [25], this is implemented by enabling interface references to hold a list of addresses rather than just a single address, and providing clients with the capability to transparently try the second address if the first one fails, and so on. This support allows centralised components in groups (e.g. the group management component) to be passively replicated without (recursively) relying on another group to provide the replication.

Within our prototype environment we have implemented the group factory, the bootstrap group management component, and a number of useful group related components. The most basic of these components is a *primitive multi-party channel*. This is a distributed binding that implements basic multi-party communication using multicast IP. It supports a single interface type through which the user can send or receive a message to/ from all the currently active interface instances. We have also implemented a *reliability* component which can be used in conjunction with the primitive channel to yield a *reliable multi-party binding*. Multiple instances of this component, one at each site, interact according to a simple SRM-like protocol [10] to deliver reliable messages over the primitive multi-party channel.

We have also developed a number of *filter* components that deal with various aspects of audio and video encoding and compression (because of our use of Python we can only deal with very low resolution video). In addition we have implemented *collators*, which are used to receive a set of messages and reduces them to one single message according to a configurable policy. These are typically used in the implementation of actively replicated servers to yield a single reply from  $n$  servers. They can also be used to select an appropriate output from different implementations of the same function in an  $n$ -redundant array.

We have experimented with a large number of group configurations, including various audio and video conferencing scenarios and both active and passive replica fault tolerant groups, by combining the above components in various ways. We are currently developing additional components to further extend the scope of the architecture. Most notably, we are developing components which implement ordering protocols such as causal and total order-

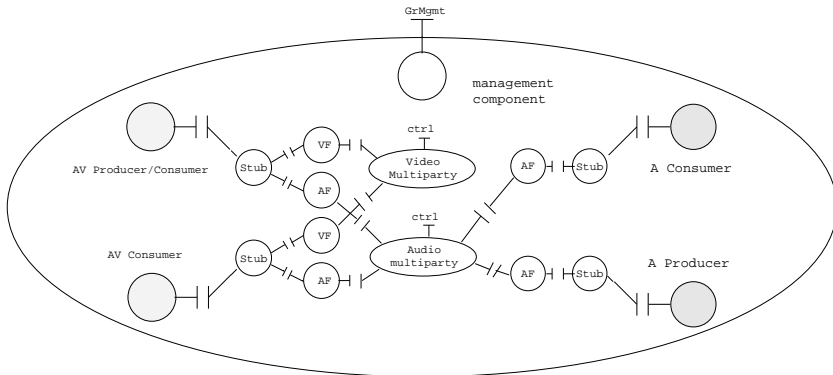
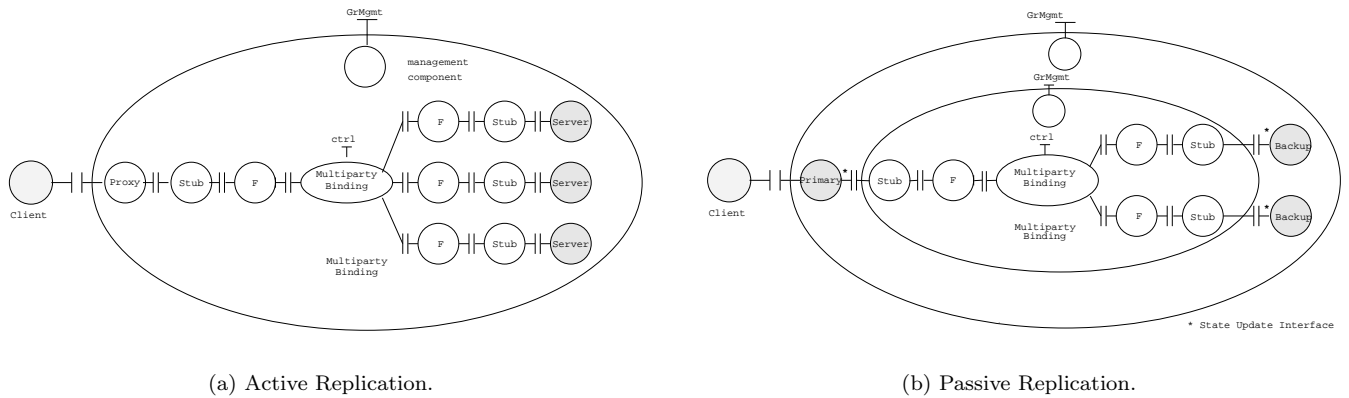


Figure 3: Example of audio/video dissemination scenario.



(a) Active Replication.

(b) Passive Replication.

Figure 4: Example of fault-tolerant scenarios.

ing [3].

## 5 Related Work

Our component model is influenced by models such as COM+ [18], Enterprise JavaBeans [30] and CORBA Components [23], 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.

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 [33], work at Michigan [15] and, more recently, the “building block” based reliable multicast protocol proposed by the IETF [35]. 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.

More recent work at Cornell on the Quintet system [34] 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 is not as radical as our reflection based approach. In addition, the work is not targeted at the full range of group applications.

In the CORBA world, the Object Management Group (OMG) has defined a multi-party event service [22] and has recently added fault tolerance by replication of objects to its specification [25]. However, as outlined in the introduc-

tion, 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 [17], Eternal [21], OGS [9], OFS [29] and NewTop Object Group Service [20] are all targeted 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 through the setting of pre-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 [19] for the Java RMI (Remote Method Invocation) and the work described in [14] in the context of the Regis platform. In addition, ARMADA [1] and Chameleon [13] are middleware systems for fault tolerance. Similar comments to the above can be applied to all these systems. ARMADA and Chameleon explicitly address the need for adaptivity, but not through a component/ reflection based approach.

Finally, significant research has been carried out on group services for multimedia and CSCW. Examples are the Group and Session Management System (GSM) [36], GroupKit [11], and the work of Dourish et al [8] and Anker et al [32]. Again, these systems target a particular area of group support and cannot naturally be applied more generally.

## 6 Conclusions

Our experience to date with the group services framework described in this paper has been very favourable. With the power and generality of the component architecture, coupled with the flexibility of the group factory, we have been able to build, with minimal effort, a wide range of group services using only a relatively limited set of base components. It may prove necessary to extend the group factory's template language in the light of further experience but it has so far proved adequate to all the scenarios we have evaluated.

In a future implementation phase we will migrate our implementation to the OpenORB v2 environment [27] which also implements the component architecture described in section 2. However, OpenORB v2, which is currently under development at Lancaster, supports components written in multiple languages and performs significantly better than the Python based prototype (it is

based on a binary level inter-component communication scheme, called OpenCOM, which is a superset of a subset of Microsoft's COM). This will permit more realistic experimentation with media dissemination groups and will also allow us to address questions regarding the performance implications of our approach.

One possible drawback of our design is the potential complexity of template definitions; particularly for large and complex groups. We are currently addressing this issue by experimenting with the notion of *front-end factories*. The idea is that different front ends will be provided for different application areas. Each front end will provide a custom interface and map this to the standard XML template accepted by the generic group factory.

A final area of future work is to investigate in detail the full power of the reflective capability of groups. This relates, for example, to the issues of reconfiguration that were touched on in section 1 but there are many further possibilities; e.g., using reflection to passivate and activate groups, insert 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 runtime [5]. However, it seems a highly desirable goal to offer such facilities.

## Acknowledgments

Katia Barbosa Saikoski would like to thank her sponsors, Federal Agency for Post-Graduate Education (CAPES), Brazil and Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil. Special thanks to Fábio Costa who has discussed many issues related to this work.

## References

- [1] T. F. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. G. Shin, Z. Wang, and H. Zou. ARMADA Middleware and Communication Services. *Real-Time Systems*, 16(2-3):127–153, May 1999.
- [2] A. Andersen. The Open-ORB Python prototype API. NORUT IT Report IT302/2-99, NORUT IT, Oct. 1999.
- [3] K. Birman. *Building Secure and Reliable Network Applications*. Manning, 1996.

- [4] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.
- [5] G. S. Blair, L. Blair, V. Issarny, and A. Z. P. Tuma. The role of software architecture in constraining adaptation in component-based middleware platforms. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware2000, IBM Palisades Executive Conference Center, Hudson River Valley (near New York City) New York, USA, 2000*.
- [6] F. M. Costa, H. A. Duran, N. Parlavantzas, K. B. Saikoski, G. Blair, and G. Coulson. The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 79–99. Springer-Verlag, Heidelberg, Germany, June 2000.
- [7] G. Coulson, G. Blair, N. Davies, P. Robin, and T. Fitzpatrick. Supporting Mobile Multimedia Applications through Adaptive Middleware. *IEEE Journal on Selected Areas in Communications*, 17(9):1651–1659, September 1999.
- [8] P. Dourish. Developing a Reflective Model of Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, 2(1):40–63, 1995.
- [9] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, Département D'Informatique – École Polytechnique Fédérale de Lausanne, 1997.
- [10] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Session and Application Level Framing. *IEEE/ACM Transactions on Networking*, December 1997.
- [11] S. Greenberg and M. Roseman. *Computer-Supported Cooperative Work (Trends in Software 7)*, chapter Groupware Toolkits for Synchronous Work. John Wiley & Sons, 1999.
- [12] ISO/IEC. Open Distributed Processing Reference Model, Part 1: Overview. ITU-T Rec. X.901 — ISO/IEC 10746-1, ISO/IEC, 1995.
- [13] Z. T. Kalbarczyk, S. Bagchi, K. Whisnant, and R. K. Iyer. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Trans. on Parallel and Distributed Systems*, 10(6), June 1999. Special Issue on "Dependable Real Time Systems.
- [14] C. Karamanolis and J. Magee. A Replication Protocol to Support Dynamically Configurable Groups of Servers. In I. C. S. Press, editor, *Third International Conference on Configurable Distributed Systems (IC-CDS'96)*, Annapolis MD, May 1996.
- [15] R. Litiu and A. Prakash. Adaptive Group Communication Services for Groupware Systems. In *Second International Enterprise Distributed Object Computing Workshop (EDOC'98)*, San Diego, CA, November 1997.
- [16] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, volume 22 of *ACM SIGPLAN Notices*, pages 147–155. ACM Press, 1987.
- [17] S. Maffeis. Adding Group Communication Fault-Tolerance to CORBA. In *Proceedings of USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995.
- [18] Microsoft Corporation. COM Home Page. Internet Publication - <http://www.microsoft.com/com/>, December 1999.
- [19] A. Montresor. The Jgroup Reliable Distributed Object Model. In *Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, Helsinki, Finland, June 1999.
- [20] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and Implementation of a CORBA Fault-tolerant Object Group Service. In *Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, Helsinki, Finland, June 1999.
- [21] P. Narasimhan, E. Moser, and P. M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering Journal*, 4(3):139–150, Sept. 1997.
- [22] Object Management Group. Event service, v1.0. Formal/97-12-11, December 1997.

- [23] Object Management Group. Corba components final submission. OMG Document orbos/99-02-05, 1999.
- [24] Object Management Group. CORBA Object Request Broker Architecture and Specification - Revision 2.3, June 1999.
- [25] Object Management Group. Fault Tolerant CORBA Specification, V1.0. <http://www.omg.org>, April 2000. OMG document: ptc/2000-04-04.
- [26] E. Oskiewicz and N. Edwards. A Model for Interface Groups. Technical Report 1002.01, APM Ltd., Poseidon House, Castle Park, Cambridge, CB3 ORD, UK, May 1994.
- [27] N. Parlavantzas, G. Coulson, and G. Blair. Applying Component Frameworks to Develop Flexible Middleware. In *ECOOP'2000 Workshop on Reflection and Meta-level Architectures*, 2000.
- [28] K. B. Saikoski, L. Johnston, G. Coulson, and G. Blair. Towards a Configurable and Reconfigurable Component-Based Group Service. Technical Report MPG-99-22, Computing Department, Lancaster University, August 1999.
- [29] G.-W. Sheu, Y.-S. Chang, D. Liang, S.-M. Yuan, and W. Lo. A Fault-Folerant Object Service on CORBA. In *17th International Conference on Distributed Computing Systems (ICDCS '97)*, Baltimore, MD, May 27-30 1997.
- [30] Sun Microsystems. Enterprise JavaBeans specification version 1.1. <http://java.sun.com/products/ejb/index.html>, 2000.
- [31] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [32] D. D. Tal Anker and I. Keidar. Fault Tolerant Video-on-Demand Services. Technical report, Computer Science Department of the Hebrew University of Jerusalem., 1999.
- [33] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building Adaptive Systems Using Ensemble. Technical Report TR97-1619, Cornell University, February 1997.
- [34] W. Vogels, D. Dumitriu, M. Pantiz, K. Chipawolski, and J. Pettis. Quintet, Tools for Reliable Enterprise Computing. <http://www.cs.cornell.edu/rdc/quintet/edoc.html>.
- [35] B. Whetten, L. Vicisano, R. Kermode, M. Handley, S. Floyd, and M. Luby. Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer. INTERNET-DRAFT - RMT Working Group, Internet Engineering Task Force, 10 March 2000. `draft-ietf-rmt-buildingblocks-02.txt`.
- [36] E. Wilde. *Group and Session Management for Collaborative Applications*. PhD thesis, Swiss Federal Institute of Technology, Zürich, 1997. Diss ETH No 12075.
- [37] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. W3C Recommendation, Feb. 1998.