The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

The Object Group Design Pattern

Silvano Maffeis
Olsen & Associates

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

The Object Group Design Pattern

Silvano Maffeis*
maffeis@acm.org

Olsen & Associates
Zurich, Switzerland

Abstract

This paper describes "Object Group", an object behavioral pattern for group communication and fault-tolerance in distributed systems. The Object Group pattern supports the implementation of replicated objects, of load sharing, and of efficient multicast communication over protocols such as IP-multicast or UDP-broadcast. Application areas of the pattern are fault-tolerant client/server systems, groupware and parallel text retrieval engines. Events within an Object Group honor the Virtual Synchrony model. With Virtual Synchrony, the size of an object group can be varied at run-time while client applications are interacting with the group. A replicated state remains consistent in spite of objects entering and leaving the group dynamically and in spite of failures. The Object Group pattern has been implemented in the Electra and in the Orbix+Isis CORBA Object Request Broker.

Keywords: Distributed Systems, Design Patterns, Replication, Group Communication, Virtual Synchrony.

1 Intent

The Object Group design pattern provides a local surrogate for a *group* of objects distributed across networked machines. The members of an object group have a consistent view of which other objects are also part of the group and are notified of objects joining and leaving the group. The pattern also provides a high-level, object oriented interface to low-level multicast facilities such as Ethernet, ATM, UDP-broadcast and IP-multicast.

*Supported by the Swiss National Science Foundation (Schweizer Nationalfonds) while on leave at Cornell University, Dept. of Computer Science.

2 Also Known As

Server Group Service, Replicated Object Pattern, Active Replica Pattern.

3 Motivation

To illustrate the object group pattern, consider the following CORBA IDL interface:

```
// IDL
interface Directory {
    void install(in string key, in any value)
       raises (ENTRY_EXISTS);
    void lookup(in string key, out any value)
       raises (NO_SUCH_ENTRY);
    void remove(in string key, out any value)
       raises (NO_SUCH_ENTRY);
};
```

The `Directory` interface could be used to implement an online phone book service. To ensure high availability of the service, several copies of the `Directory` are created on different hosts and are joined to an object group. Client applications use one object reference to represent the whole group without being concerned about the references of the individual group members.

The `remove` and `install` operations are multicast to the group because these modify the replicated state of the directory. A `lookup` operation, on the other hand, needs to be directed to only one group member as the state of the service remains unaltered.

Each group member replies to a `remove` operation. However, the client's runtime system returns only the first arriving member reply (transparent multicast style). If required, clients can access the replies of all group members using a non-transparent multicast style.

An object group request succeeds as long as at least one group member survives the request. The replication degree of an object can be increased at run-time. When an object requests to join a group, the runtime system first suspends group invocations. Then the runtime system requests the internal state[1] of a group member by issuing its `get_state` member function. Subsequently, the runtime transfers the state over the network to the newcomer object and invokes its `set_state` function with the state as a parameter. Finally, group invocations are resumed. The whole procedure is fault-tolerant; if the state-sending member fails, state transfer is restarted with another member.

The underlying group management protocol guarantees a consistent replicated state of the `Directory` even when several clients fire `install` and `remove` operations while a new object is joining the group. This execution style is called *Virtual Synchrony* [2].

4 Applicability

The object group pattern can be very effective when there is a need for fault-tolerance, efficient dissemination of data, load-sharing or a combination thereof. Some interesting application areas are:

- **Fault-tolerant client/server systems**. The proposed pattern supports the implementation of highly available objects. Object groups can provide active replication, passive replication, and multi-versioning.

- **Groupware**. Object groups facilitate the implementation of applications such as teleconferencing, video-on-demand, distributed whiteboards and other kinds of groupware. Information flows between groups of individuals can be modeled and implemented in an efficient manner.

- **Ticker services for financial information**. In such applications, a continuous data stream (for example quotes from a stock exchange) needs to be transmitted to a possibly large set of receivers (for example to trader workstations) in an efficient and reliable manner (Figure 1). The pattern allows the efficient transmission of data over protocols such as IP-multicast. Reliability is ensured by object replication and reliable multicast.

- **Scalable text retrieval systems**. A parallel text retrieval engine can be implemented in the form of an object group with N members (Figure 2). Each group member is assigned a portion of the text database. To perform a search, a client application multicasts the sought text pattern to the group. Each of the N group members needs to search only 1/N of the database. As is explained below, group members have a consistent opinion on the group size in spite of failures and in spite of retriever-objects that join the group at run-time to increase parallelism. For this kind of application, a nearly linear speedup can be achieved by employing the object group pattern.

- **Caching**. In certain situations, the response time of a service is decreased if provided by an object group since replicas of the service can be placed at the sites where the service is frequently accessed. In the extreme case, a replica is created per client application and placed on the clients' machines. A client transmits read-only requests only to the local replica; write-requests, on the other hand, are multicast to all replicas.

- **Network management**. Object groups offer a convenient solution to the problem of collecting and propagating management and monitoring information in distributed applications.

- **Object migration**. Requests are multicast through opaque object references and senders need not know the network addresses of the members. Consequently, an object can leave a group, move to another place and then rejoin the group and continue working. Object groups hence offer support for mobility and for system reconfiguration. This kind of object migration is useful for preventive maintenance and for coarse grained load balancing.

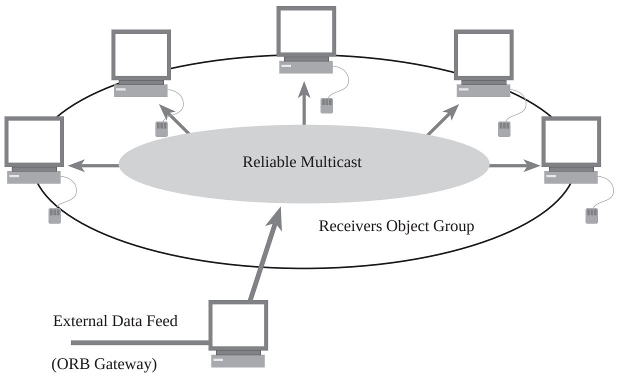

Figure 1: Reliable Stock Exchange Ticker.

5 Virtual Synchrony

The main requirement our pattern makes on the underlying communication subsystem is that it imple-

[1] for example, the entries in an electronic phone book.

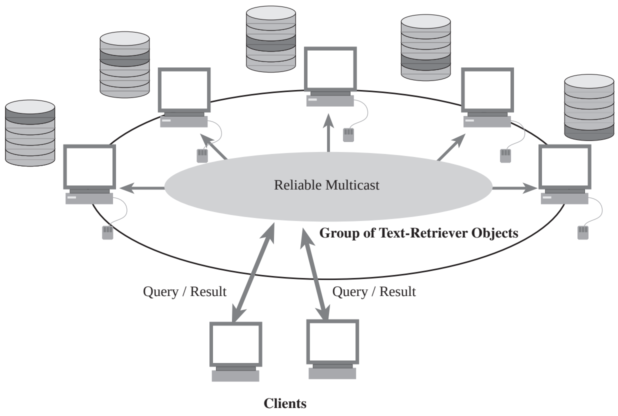

Figure 2: Scalable Text Retrieval Engine.

ments the Virtual Synchrony model. Virtual Synchrony makes possible the run-time replication of mutable objects and the consistent detection of failures. Moreover, the notion of consistent management of group membership information is a fundamental component of the Virtual Synchrony model. The motivation behind Virtual Synchrony is to allow programmers to assume a closely synchronized style of distributed execution, even though the underlying system is asynchronous [2]:

- The execution of an object consists of a sequence of events which may be internal computations, request transmissions, request dispatch, failure notifications or changes to the membership of groups of which the object is a member.
- A global execution of the system consists of a set of object executions. At the global level requests are sent by multicast or unicast.
- Any two objects that receive the same multicasts or observe the same group membership changes see the corresponding local events in the same relative order.
- A multicast to an object group is delivered to its full membership. The send and delivery events appear as a single, instantaneous event.
- However, the model provides only the *illusion* of synchronous execution: Whenever possible, synchronization is relaxed to reduce the degree to which processes can delay one another, and to better exploit the parallelism inherent to an application.

6 Structure and Participants

Figure 3 depicts structure and participants of the object group pattern. Figure 4 shows a possible layering of the communication subsystem.

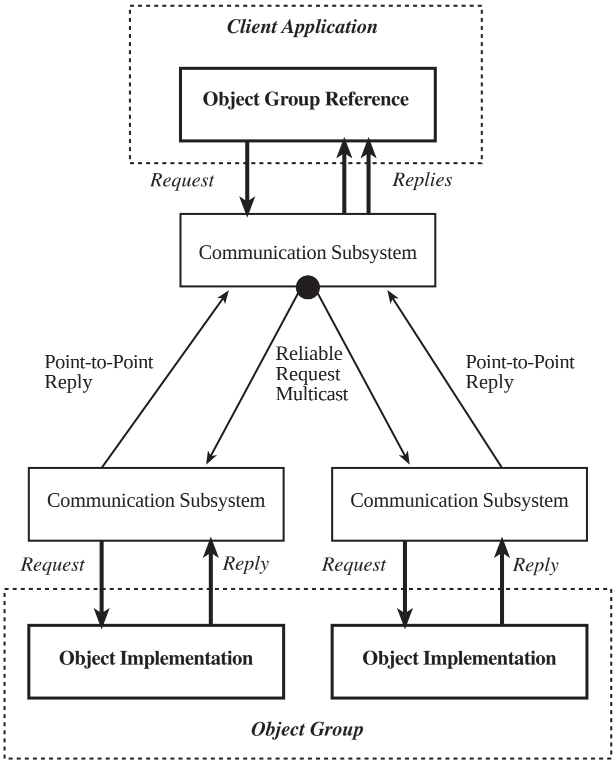

Figure 3: Structure of the Object Group Pattern.

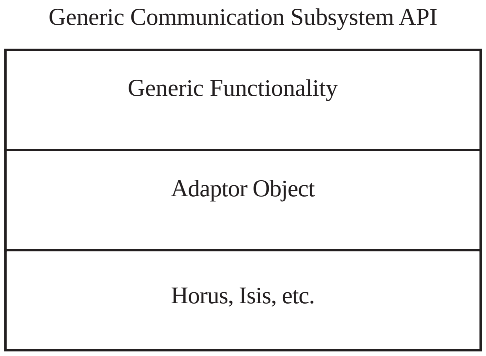

Figure 4: Layers of the Communication Subsystem.

Client Application

The *client application* creates an instance of an object group reference and binds the reference to a target group by using a name server to perform the name-to-object mapping. After successful binding, operations issued through the reference are transmitted to the object group.

Object Group Reference

An *object group reference* is used as a "smart pointer" to an object group. The object group reference is a surrogate object with the same interface as the group members. When a member function is invoked on the group reference, the runtime marshals the arguments, multicasts the request to the group members, waits for one or more member-replies, unmarshals the replies and passes the result back to the client. An object group reference can be used like a conventional CORBA object reference.

Communication Subsystem

The *communication subsystem* (or *runtime*) provides a low-level interface to reliable multicast, group management and light-weight processes. In this paper, we assume that the communication subsystem is part of a CORBA object request broker. We suggest implementing the communication subsystem on top of toolkits such as Horus [19], Isis [2], Consul [12], Phoenix [11], Totem [13], or Transis [1] as these provide low-level system support for process groups, reliable multicast and Virtual Synchrony. For the sake of flexibility and portability, the communication subsystem implements a generic interface. An Adaptor Object is used to map the generic interface onto the real interface provided by the toolkit (Figure 4).

Object Implementation

An object group consists of one or more *object implementations*. An object implementation provides the functionality of the service; for example, it implements a phone directory or a receiver of stock exchange quotes. Each group member implements the same IDL interface.

Object Group

The object implementations form a logical *object group*. The ORB ensures that each group member receives all requests that the client applications have submitted through the object references (reliable multicast). Optionally, the ORB may guarantee that each object implementation dispatches requests in exactly the same order (totally ordered multicast).

Each group member obtains a unique *rank* number. The oldest member is assigned rank 0, the next-oldest obtains rank 1 and so forth. Whenever an object joins or leaves (either explicitly or by crashing). the group, the group members obtain a *view change notification* from the ORB to inform them of their new ranks and of the number of group members.

7 Collaborations

Server Side

If an object implementation wants to join a group, it obtains an object group reference from a name server or by using the CORBA `string_to_object` function. Subsequently, it invokes a `join` function with the reference as a parameter. (There is also a `leave` and a `create_group` function). After calling `join`, its `set_state` member function is invoked by the ORB to update its internal state to the state of the other group members. In the above `Directory` example, the state consists of the `string/any` entries maintained by the service.

Client Side

When an application wants to send a request to an object group, it obtains an object group reference either from a name server or by using the CORBA `string_to_object` function. Now the application can submit requests through the group reference in a type-safe manner. If required, the client application can request that each group member's reply be returned (non-transparent multicast).

View Management

At all times, a member of a group has a certain *view* of which other objects belong to its group. A view is an ordered set of object references; there is one reference per group member. The first entry of a view (`view[0]`) represents the oldest member, the second entry (`view[1]`) the second oldest and so forth. Each group member knows its own index in the view. This ranking is consistent among all group members. Join and leave operations trigger the installation of a new view at each group member's runtime system.

View management is an important component of the Virtual Synchrony model. To hold views consistent, the object group pattern employs a view management protocol similar to the following (Figure 5).

The oldest member is said to be the *coordinator* of the group. When an object wishes to join a group, the object's runtime sends a `JOIN_REQUEST` message to `view[0]` (to the coordinator). The coordinator multicasts a `FLUSH_REQUEST` message to the group members. Upon receipt of a `FLUSH_REQUEST`, a member

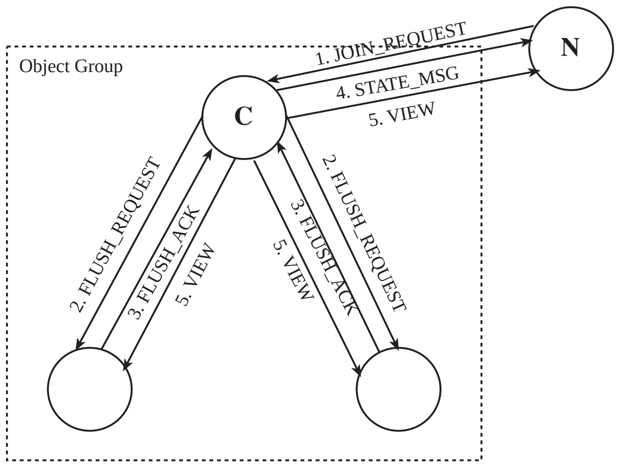

Figure 5: View Management Protocol. Object N requests to join a group containing three objects. Object C is the coordinator of the group.

submits any cached requests it needs to have delivered in the old view. Each member confirms that it is done with the flushing by returning a `FLUSH_ACK` message to the coordinator. A member is not allowed to submit any more requests since the view is now unstable. When the coordinator has received a `FLUSH_ACK` from every surviving member, it sends one or several `STATE_MSG`s containing its internal state to the newcomer object. After this state transfer is completed, the coordinator multicasts a `VIEW` message containing the new view. When a member receives a `VIEW` message, it updates its local `view` vector with the data in the message. The view is stable and the members may continue to submit requests.

If the coordinator fails, the group members receive a failure notification from the runtime and the member with object reference `view[1]` becomes the new coordinator. The view transfer is then restarted with the new coordinator.

The view management mechanisms are mostly transparent to the user of the object group pattern. However, by overloading specific member functions, an application dependent action can be triggered whenever a new view is installed. Variations of this view management protocol are discussed in [17, 15, 7, 4].

8 Consequences

The object group pattern offers the following main benefits:

- *Fault-tolerance* through active replication, passive replication or multi-versioning. This makes possible fault-tolerant client/server systems.
- *Efficient group communication* over Ethernet, IP-multicast and other facilities that enable multicast. The object group pattern provides a high-level, invariable interface to such facilities. This supports the implementation of groupware systems and of certain applications in the financial domain.
- *Load sharing* through a group of objects in which each member performs only a part of the computations associated with a client's requests. This makes possible parallel text retrieval applications and the like.
- *Scalability* through the replication of services which are mainly accessed by read-only operations.
- *System reconfiguration* through dynamic object migration.
- Last but not least, an object-oriented interface to toolkits like Horus and Isis is provided.

The object group pattern has following drawbacks:

- An efficient and robust implementation of the pattern requires *sophisticated system support* not yet available in today's commonly used operating systems. This system support includes: reliable, totally ordered multicast of network messages, a group management protocol to ensure that object group members have consistent views on which objects are in the group, failure detection and consistent propagation of failure notifications.
- Fault-tolerant *name servers* must be provided, allowing objects to retrieve the references of the groups they want to join.
- Fault-tolerance is *not transparent* to the programmer. Programmers have to deal with object groups, with join and with leave requests. For stateful objects, programmers need implement state transfer upcalls unless an IDL compiler is provided to generate state-transfer code out of enriched IDL specifications.

9 Implementation

Group Management API

In a CORBA Object Request Broker, the object group pattern can be implemented *as an extension* of the CORBA Basic Object Adapter (BOA):

```
// C++
class GroupBOA: virtual public BOA {
public:
    // Operations on object groups:
    //
    static void create_group(Object_ptr group,
        const ProtocolPolicy& policy
        =default_protocol_policy);
    void join(Object_ptr group);
    void leave(Object_ptr group);
    static void destroy_group(Object_ptr group);

    virtual void get_state(AnySeq& state,
        Boolean& done);
    virtual void set_state(const AnySeq& state,
        Boolean done);
    virtual void view_change(const View& newView);
};
```

The `create_group` method creates a new object group and binds the object reference `group` to it. The reference can be installed in a name server or converted to a human-readable string by the `ORB::object_to_string` operation. The `policy` argument tells the underlying toolkit what kind of multicast protocol to employ; for example, total ordering or causal ordering [8, 5]. If the BOA is configured to run on Isis, the `policy` object selects one of the Isis *abcast*, *cbcast*, *fbcast* or *gbcast* protocols. In the Horus configuration, the `policy` object selects a Horus protocol stack[2] [18] . The policy-object mechanism can be extended to cover quality of service guarantees. For example, the minimum bandwidth necessary to sustain a certain service can be defined through a policy object.

Objects in the network join or leave a group by retrieving its reference from a name server and by issuing the `join` or `leave` operation with the reference as a parameter. `destroy_group` irrevocably destroys an object group but without destroying the group-members themselves.

State Transfer

When an object joins a non-empty group, the ORB requests the internal state (that is, the values of the instance variables) of some group-member by invoking its `get_state` method. Subsequently, the runtime transfers the state to the newcomer and invokes the newcomer's `set_state` method. The runtime can transfer a large state in fragments by repeatedly invoking the state transfer functions until `TRUE` is assigned to the `done` argument of `get_state`. An object-state is represented as a sequence of CORBA `any` objects.

[2]Horus supports a variety of ordering and reliability protocols, as well as communication over UDP, IP-multicast, ATM, Mach messages, and so forth.

State transfer is necessary for redundant computations to permit the replication-degree of a stateful object to be increased at run-time. The programmer must write application-specific `get_state` and `set_state` methods. These methods can also be used to checkpoint the state of an object to non-volatile storage or to perform object migration.

The `view_change` method of an object is invoked whenever another object joins or leaves the group. The `newView` object contains the new cardinality of the group as well as the object references of the group members.

Call Styles

The programmer may specify how many member-replies the runtime should collect during an object-group invocation. Furthermore, operations that leave the state of an object unchanged can be tagged as *readonly*. Such operations are automatically transmitted by point-to-point communication to only one group member. To help implement this, we suggest that a `readonly` operation attribute be provided by the IDL. Following call styles can be selected by the programmer on a per-call basis:

- **ALL**: This call type demands that the replies of all operational group-members be collected and returned to the caller. The caller is suspended until all group members have replied.
- **MAJORITY**: The call is active only until a majority of the members have replied.
- **ONE**: The call is active only until the first member-reply is received. This call-style is used for transparent multicast.
- N: Any number ranging rom 1 to the number of members of the group can be specified. If the specified number of replies cannot be collected due to a failure, an exception is thrown.

Object Migration

In order to migrate an object implementation that is the only member of a certain group, one has to instantiate a new object implementation on the destination machine and to join the object to the group. The state of the obsolete object will automatically be transferred to the newcomer object and the two objects will be synchronized. The obsolete object can then be destroyed.

10 Sample Code

The following code illustrates the replication of a `Directory` object and the interaction with the resulting `Directory` object group.

Server Side

The code fragment below demonstrates a server application that instantiates an implementation of interface `Directory`, creates an object group if given the `-c` option, and joins the newly created object to the group. The `-c` option also causes a group reference to be installed into the naming service. The `-j` option causes the `Directory` object implementation to join an existing group.

```
// C++
main(int argc, char **argv) {
    Object_var obj, group;
    // a reference to the naming service:
    extern CosNaming::NamingContext_ptr nc;
    // create an implementation of interface Directory:
    _im_directory obj0;
    ...
    switch(argv[1][1]){
    case 'c': // create and join a group:
        // get a reference for obj0:
        obj = obj0.create(rd, pt, dpt);
        // create an empty object group:
        obj0.create_group(obj);
        // install group reference into name server:
        nc->bind("directory", obj);
        // join obj0 to group:
        obj0.join(obj);
        break;
    case 'j': // join an existing group:
        // retrieve group reference from name server:
        group = nc->resolve("directory");
        // join obj0 to group:
        obj0.join(group);
        break;
    default:
        cerr << "usage: " << argv[0] << " -c|-j\n";
        exit(1);
    };
}
```

Client Side

The following code fragment shows a procedure that is part of a client application. The object group reference is retrieved from the name server. Subsequently, a transparent and a non-transparent request is issued through the reference. A non-transparent request is appropriate when the members of a `Directory` group provide different information; for example, one member might maintain business phone numbers, another member home numbers:

```
// C++
void clientProc(){
    extern CosNaming::NamingContext_ptr nc;
    Any entry; AnySeq *entries;
    char *number;
    Environment env; EnvironmentSeq envs;
    directory_var dir =
        Directory::_narrow(nc->resolve("directory"));

    // transparent multicast
    dir->lookup("J. Smith", entry, env);
    entry >>= number;
    cout << "The phone number of J. Smith is "
        << number << "\n";
    ...

    // non-transparent multicast
    dir->lookup("J. Smith", entries, envs);
    for(i =0; i < entries->length(); i++){
        (*entries)[i] >>= number;
        cout << "A phone number of J. Smith is "
            << number << "\n";
    };
    delete entries;
}
```

11 Known Uses

The object group pattern has been implemented in the Electra [9, 10] and in the Orbix+Isis [6] ORB.

Electra

Electra is a flexible CORBA-2 Object Request Broker based on the object group paradigm. In Electra, the object group pattern is implemented as an extension of the CORBA Basic Object Adapter (BOA), as was proposed in Section 9. Electra supports the dynamic replication of CORBA objects, failure detection, asynchronous communication and light-weight processes. The ORB is portable; the present version of Electra runs on both Horus and Isis. Electra is publicly available through the Web at `http://www.olsen.ch/~maffeis/electra.html`.

Orbix+Isis

Orbix+Isis is an adaptation of Orbix from Iona Ltd. that runs on top of the Isis toolkit. To become a member of an object group, an object implementation must inherit from a base class that implements either an Active Replica or an Event Stream group style. The Active Replica object group offers three communication styles: Multicast, Client's Choice, and Coordinator/Cohort. Orbix+Isis is a commercial product available from Isis Inc. For more information contact `info@isis.com`.

12 Related Patterns

Proxy Pattern

A *Proxy* provides a surrogate or placeholder for another object to control access to it [16, 3]. Analogously, an *Object Group Reference* provides a surrogate or placeholder for a group of objects.

Coordinator/Cohort Pattern

Coordinator/Cohort [2, 10] is a form of redundant computation in which only one object (the coordinator) performs the computations associated with the client requests. Several cohort objects are associated with a coordinator, acting as "hot standbys". When the coordinator fails, one of its cohorts takes over.

Coordinator/Cohort is a special form of the object group pattern: coordinator and cohorts make up an object group in which the oldest group member is the coordinator. When the coordinator fails, the cohorts receive a view change notification from the ORB and the oldest cohort becomes the new coordinator. New cohorts can be included at run-time.

Event Channel Pattern

The *Event Channel Pattern* [14] provides the abstraction of a highly available, persistent message bus. The Event Channel allows an object to "post" requests and to "subscribe" for requests it is interested in[3]. Posting and subscription can be by ASCII strings that represent topics of interest.

Objects can post requests for receivers that happen to be unavailable. To that purpose, the Event Channel provides persistency of requests. Receivers may later connect to an Event Channel to retrieve a backlog of requests. Thus, clients are decoupled from servers and communication is asynchronous.

The Event Channel pattern can be seen as a variation of the object group pattern. The patterns differ mainly in that Event Channel allows its members to *selectively* listen only for certain requests. Another difference is that Event Channels allow the *spooling* of requests on non-volatile storage so that requests can be directed to group members that are temporarily unavailable. Object Group, on the other hand, excludes any member that has been unresponsive for a relatively short period of time, typically in the range of 10 to 30 seconds.

[3] similar to Usenet.

Acknowledgements

The author would like to thank Robert Ward for many helpful comments. A preliminary version of this material was discussed at the OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems. The author would like to thank the workshop participants for their valuable suggestions and for their encouragement.

References

[1] Amir, Y., Dolev, D., Kramer, S., and Malki, D. Transis: A Communication Sub-System for High Availability. In *22nd International Symposium on Fault-Tolerant Computing* (July 1992), IEEE.

[2] Birman, K. P., and van Renesse, R., Eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[4] Golding, R. A., and Taylor, K. Group Membership in the Epidemic Style. Tech. rep., University of California, Santa Cruz, 1992.

[5] Hadzilacos, V., and Toueg, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993.

[6] Isis Distributed Systems, Inc., Iona Technologies, Ltd. *Orbix+Isis Programmer's Guide*, 1995. Document D071-00.

[7] Jahanian, F., Fakhouri, S., and Rajkumar, R. Processor Group Membership Protocols: Specification, Design and Implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems* (Princeton, New Jersey, Oct. 1993), IEEE.

[8] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978).

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

[10] Maffeis, S. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, University of Zurich, Department of Computer Science, 1995.

[11] Malloth, C. P., Felber, P., Schiper, A., and Wilhelm, U. Phoenix: A Toolkit for Building Fault-Tolerant, Distributed Applications in Large Scale. In *IEEE SPDP-7 Workshop on Parallel and Distributed Platforms in Industrial Products* (San Antonio, TX, Oct. 1995), IEEE.

[12] Mishra, S., Peterson, L. L., and Schlichting, R. D. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal 1*, 2 (Dec. 1993).

[13] Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., Budhia, R. K., Lingley-Papadopoulos, C. A., and Archambault, T. P. The Totem System. In *Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing* (Pasadena, CA, June 1995).

[14] Object Management Group. *Common Object Services Specification Volume I.* OMG Document 94-1-1.

[15] Ricciardi, A. M. *The Group Membership Problem in Asynchronous Systems.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Nov. 1992. No. 92-1313.

[16] Shapiro, M., et al. SOS: An Object-oriented Operating System – Assessment and Perspectives. *Computing Systems 2*, 4 (Dec. 1989).

[17] van Renesse, R. The Horus Uniform Group Interface. Horus Documentation.

[18] van Renesse, R., and Birman, K. P. Protocol Composition in Horus. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario Canada, Aug. 1995).

[19] van Renesse, R., Birman, K. P., and Maffeis, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (Apr. 1996).