Annotation Markers for Runtime Replication Protocol Selection

Hein Meling

Department of Electrical Engineering and Computer Science,
University of Stavanger, N-4036 Stavanger, Norway
hein.meling@uis.no

Abstract. This paper presents an architecture enabling developers to easily and flexibly assign replication protocols simply by *annotating* individual server methods. This avoids using costly replication protocols for all object methods, e.g. read-only methods can use less costly protocols, reserving the costly replication protocols for update methods. The architecture has been implemented in the Jgroup/ARM middleware, and enables addition of new replication protocols without modifying the core toolkit. It also supports runtime selection of replication protocol for individual methods. This can be used to support self-optimization of protocol selection by optimizing for the most appropriate configuration under a given system load.

1 Introduction

Middleware for building dependable distributed applications often provide a collection of *replication protocols* supporting varying degrees of consistency. Typically, providing strong consistency requires costly replication protocols, while weaker consistency often can be achieved with less costly protocols. Hence, there is a tradeoff between cost and consistency involved in the decision of which replication protocol to use for a particular server. But, perhaps more important is the behavioral aspects of the server. For instance, the server may be intrinsically non-deterministic in its behavior, which consequently rules out several replication protocols from consideration, e.g. atomic multicast.

This paper presents an architecture for Jgroup/ARM [11] enabling software developers to easily and flexibly select their replication protocol of choice for each individual server method. The principal motivation for the architecture is to improve the flexibility in choice of replication protocols, so as to reduce the resource consumption of dependable applications as much as possible. In many fault-tolerant systems, different replication protocols are supported at the *object level* [14, 15], meaning that all the methods of a particular object must use the same replication protocol. Jgroup [11] takes a different approach: when implementing a dependable service, the invocation semantics of each individual method can be specified separately using Java annotations [2, Ch.15]. This allows for greater flexibility as various methods may need different semantics. Hence, developers may select the appropriate invocation semantics at the *method level*, and even provide different implementations with alternative semantics. The presented architecture makes it very easy to add new replication protocols to the system, with no changes to the core toolkit. Protocol implementations are picked up automatically.

The current implementation supports four different replication protocols, or invocation semantics: *anycast*, *reliable multicast*, *atomic multicast* and *leadercast*. The latter is a variant of passive replication and permits servers with non-deterministic behavior, whereas atomic multicast can be viewed as a kind of active replication, and hence does not tolerate servers being non-deterministic. The architecture can also accommodate adaptive or runtime protocol selection based on runtime changes in the environment. A common example in which application semantic knowledge can be exploited is a replicated database with read and write methods. Often a simple *Read-One, Write-All* (ROWA) replication protocol [17] can then be used and still preserve consistency. A ROWA replication protocol can easily be implemented using anycast for read methods and either multicast, atomic, or leadercast for write methods. On the other hand, replication protocols which operate at the object level require that also simple read-only methods use the *strongest* replication protocol required by the object to preserve consistency. However, assigning appropriate invocation semantics to the methods of a server do require careful consideration to ensure preservation of consistency as well as reducing the resource consumption needed. Hence, a guideline is provided in [11], based on [8]. For example, if two methods of the same server modify intersecting parts of the shared state, they should use the same replication protocol.

By exploiting knowledge about the semantics of distributed objects, the choice of which replication protocols to use for the various methods can be used to obtain a performance gain over the traditional object level approach. Similar ideas were proposed by Garcia-Molina [9] to exploit semantic knowledge of the application to allow nonserializable schedules that preserve consistency to be executed in parallel as a means to improve the performance for distributed database systems. OGS [6, 7] also allows each method of a server to be associated with different replication protocols, but this must be explicitly encoded for each method through an intricate initialization step. The approach presented herein is much easier to use as it exploits Java annotations to mark methods with the desired replication protocol. The Spread [1] message-based group communication system can also be used to exploit semantic knowledge, since each message can be assigned a different replication protocol. JavaGroups [3] on the other hand would have required separate channels for each replication protocol. Unlike Jgroup however, neither of these two systems are aimed at RMI based systems.

Organization: In Section 2 the architecture is presented, while in Section 3 the protocol selection mechanism is covered. The leadercast replication protocol is covered in Section 4, and Section 5 covers the atomic replication protocol. Finally, Section 6 discusses potential enhancements to the architecture that would enable support for adaptive runtime selection of protocols.

2 The EGMI Architecture

The external group method invocation (EGMI) architecture of Jgroup/ARM [11] aims to provide: (a) flexibility and efficiency using to a customized RMI layer; (b) flexibility to add new replication protocols; (c) runtime adaptive selection of replication protocol (Section 6); (d) improved client-side view updating (covered in [12]).

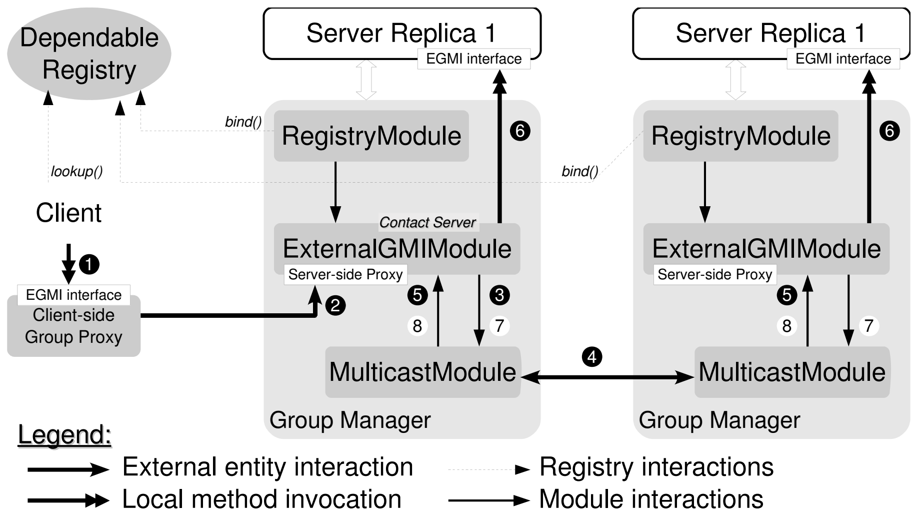


Fig. 1. The external GMI architecture

Fig. 1 illustrates the high-level interactions of the EGMI architecture. The figure illustrates interactions involved in a multicast invocation. Clients communicate with an object group through a *two-step* approach, except for the anycast semantic. Two communication steps are required for multicast interactions. The ExternalGMIModule acts as the *server-side proxy* (representative) for clients communicating with the object group, and is also responsible for protocol selection. The server representing the group is called the *contact server*. The choice of contact server is made (on a per invocation basis) by the *client-side proxy*, and different strategies can be implemented depending on the requirements of the replication protocol being used. The general strategy used by both anycast and multicast is to choose the contact server arbitrarily, while leadercast always selects the group leader. However, in the presence of failures an arbitrary server in the group is selected.

As shown in Fig. 1, before a client can invoke the object group, each member of the group must bind() its reference (client-side proxy) in the dependable registry. The client can then perform a lookup() to obtain the client-side proxy encompassing all group members. The client-side proxy provides the same EGMI interface as the server, enabling the client to invoke local methods on it (❶). The proxy encodes invocations into remote communications (❷), and ultimately complete the invocation by returning a result to the client. The ExternalGMIModule exploits the MulticastModule to send multicast messages (❸,❹,❺) to all group members. This is followed by the invocation of the encoded method (❻) on all members, and returning the results back to the contact server (⑦,⑧). The contact server is responsible for returning a selected result back to the client.

2.1 The Client-Side and Server-Side Proxies

The client-side and server-side proxies are implemented as a customized version of the Jini Extensible Remote Invocation (JERI) protocol stack [16]. All layers in JERI

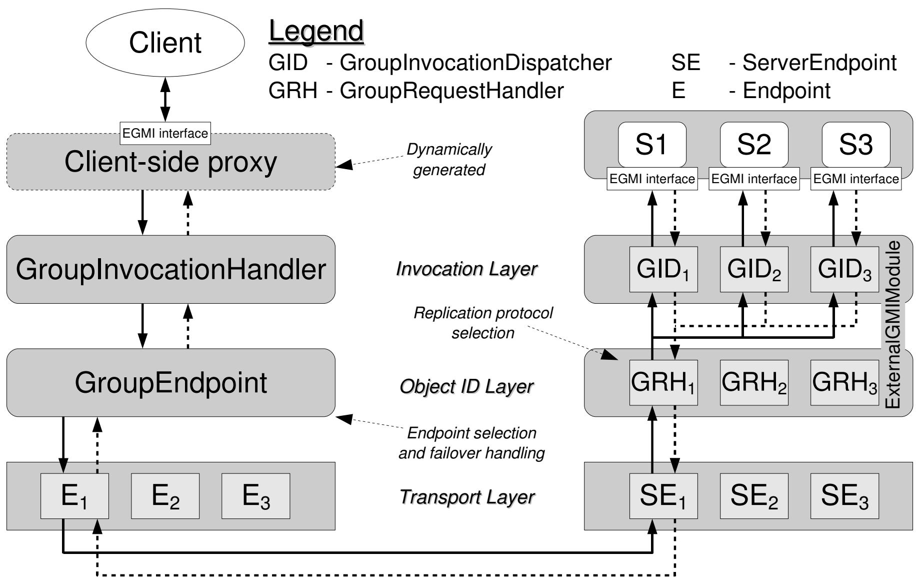


Fig. 2. The EGMI protocol stack

protocol stack have been retrofitted with group support, except for the transport layer, as shown in Fig. 2. Currently, a TCP transport is used between clients and the contact server, whereas multicast is used internally in the group.

The GroupInvocationHandler shown in Fig. 2 is responsible for marshalling and unmarshalling invocations. When invoked by the client-side proxy, internal tables are queried to determine the semantics of the method being invoked. Knowing the semantics on the client-side improves efficiency, as the contact server can forward the invocation to the group without unmarshalling it until received by the GroupInvocationDispatcher at the destination server.

The GroupEndpoint maintains the current group membership lazily synchronized with the server-side membership [12]; it stores a single Endpoint for each member of the group. Each Endpoint object represents the transport between the client and the corresponding ServerEndpoint. GroupEndpoint also selects the endpoint to use for a particular invocation, based on the semantics of the method.

When the GroupRequestHandler (GRH) receives an invocation, the invocation semantic is extracted from the data stream. Depending on the invocation semantic, the invocation is passed on to a protocol-specific invocation dispatcher (see Section 3). Here the protocol dispatcher is assumed to be multicast (as in Fig. 2). Hence, the stream is passed on to the MulticastModule, and finally to the GroupInvocationDispatcher (GID) which takes care of the unmarshalling and invocation of the method on the remote server objects. As Fig. 2 shows, the results are returned to the contact server, which finally returns the result(s) to the client.

Listing 1. Skeleton of the RegistryImpl

```
public final class RegistryImpl {
  @Multicast IID bind(String name, Entry e)
   throws RemoteException
  @Anycast Remote lookup(String serviceName)
   throws RemoteException, NotBoundException
}
```

3 Replication Protocol Selection

Each method is usually assigned a distinct invocation semantic by the server developer at design time, by prefixing each method with an annotation marker for the replication protocol to use, as shown in Listing 1. It is also possible to declare protocol annotations in the interface. However, markers declared in the server implementation takes precedence over those declared in the interface. This makes it easy to provide alternative implementations of the same interface with different invocation semantics for the various methods declared in the interface, e.g. if an implementation wants to provide stronger consistency for some methods.

Fig. 3 depicts the protocol selection mechanism of the ExternalGMIModule. Each protocol must implement the ProtocolDispatcher interface through which invocations are passed before they are unmarshalled. This allows the protocol to multicast the invocation to the other group members before unmarshalling is done in the GroupInvocationDispatcher. However, the stream received by the GroupRequestHandler is partially

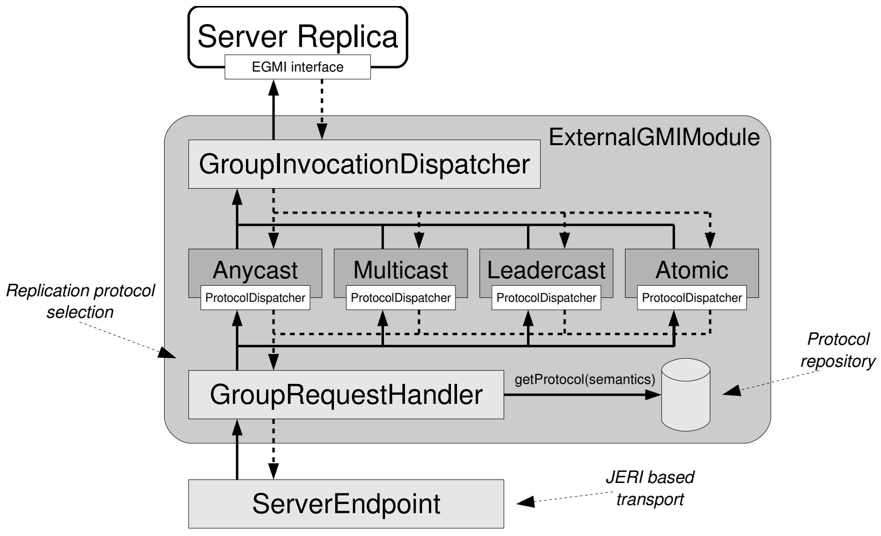


Fig. 3. EGMI replication protocol selection

Listing 2. The @Atomic annotation marker

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Atomic { }
```

Listing 3. The ProtocolDispatcher interface

```
public interface ProtocolDispatcher {
  InvocationResult dispatch(InputStream in)
    throws IOException;
  void addListener(Object listener);
}
```

unmarshalled to obtain information necessary to *route* the message to the appropriate protocol dispatcher instance. The *protocol repository* holds a mapping between the annotation marker (a method's invocation semantic) and the actual protocol instance. The repository is queried for each invocation of a method.

3.1 Supporting a New Protocol

To support new replication protocols, two additions are required: (i) a new annotation marker must be added, allowing servers to specify the new protocol and (ii) the actual protocol implementation. Listing 2 shows the annotation marker for the @Atomic replication protocol. To support runtime protocol selection, the retention policy of the marker must be set to RUNTIME to allow reflective access to the marker. Furthermore, the target element type is set so that the marker only applies to METHOD element types. For details about the Java annotation mechanism see [2, Ch.15].

A new protocol implementation must implement the ProtocolDispatcher interface (see Listing 3), and placed in the protocol package location. The latter is configured using a Java system property. Replication protocols are constructed on-demand based on reflective [2, Ch.16] analysis of the server implementation (or its EGMI interfaces) to determine the invocation semantics of its methods. Methods whose invocation semantic is unspecified defaults to @Anycast. Only required protocols are constructed. This analysis is done in the bootstrap phase, and the information is kept in internal tables for fast access during invocations.

3.2 Concurrency Issues

Note that a protocol instance may be invoked concurrently by multiple clients, and care should be taken when developing a replication protocol to ensure that access to protocol state is synchronized. Furthermore, the EGMI architecture is designed for multithreading, and hence it does not block concurrent invocations using the same or different protocols. It is the responsibility of the server developer to ensure that access to server state is synchronized. However, invocations received while a new view is pending are blocked temporarily and delivered in the next view. This is necessary to avoid that invocations modify the server state while the state merge service [13] is active.

4 The Leadercast Protocol

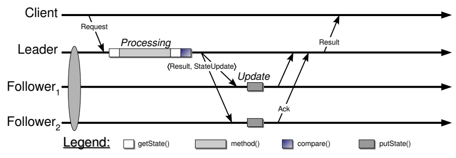
The leadercast protocol presented in this section is a variant of the passive replication protocol [10]. The principal motivation to provide this protocol is the need for a strong consistency protocol that is able to tolerate non-deterministic operations. The main difference between leadercast and the passive replication protocols described in the literature [10] is optimizations in scenarios where the leader has crashed. That is how to convey information about the new leader to clients, and how to handle failover. These optimizations are possible due to the client-side view updating technique described in [12]. Fig. 4(a) illustrates the leadercast protocol, when the client knows which of the group members is the leader. In this case, the protocol is as follows:

1. The client sends its request to the group leader.
2. The leader process the request, updating its state.
3. The leader then multicasts an update message containing $\langle$Result, StateUpdate$\rangle$ to the followers (backups).
4. The followers modify their state upon receiving an update message, and replies with an Ack to the leader.
5. Only when the leader has received an Ack from all live follower replicas, will it return the Result to the client.

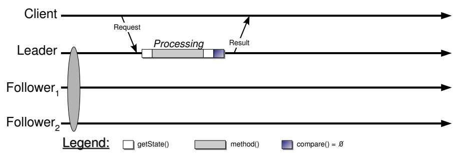
Result is the result of the processing performed by the leader, while StateUpdate is the state (or a partial state) of the leader replica after the processing. A partial state may for instance be the portions of the state that have been modified by the leadercast methods. Notice the compare() method performed at the end of the processing. This is used to compare the server state before and after the invocation of method(), and if the state did not change, there is no need to send the update message, as shown in Fig. 4(b).

The Result part of the update message is necessary in case a follower is promoted to leader, and needs to emit the Result to the client in response to a reinvocation of the same method. This can only happen if the leader fails, causing the client to perform a failover by reinvoking the method on another group member, as shown in Fig. 4(c). Hence, the followers needs to keep track of the result of the previous invocation made by clients. A result value can be discarded when a new invocation from the same client is made, or after some reasonable time longer than the period needed by the client to reinvoke the method. As depicted in Fig. 4(c), the failure of the leader causes the membership service to install a new view. Client invocations may be received before the new view is installed, however, they will be delayed until after the view has been installed, as discussed in Section 3.2. The follower receiving the reinvocation of a previously invoked method will simply return the result to the client along with information about the new leader.

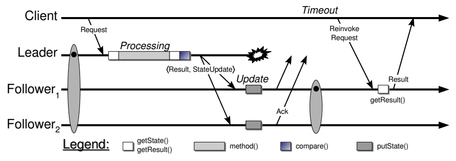
If the follower receiving a reinvocation of a previously invoked method is not the new leader, the invocation is forwarded to the current leader, as shown in Fig. 4(d). This



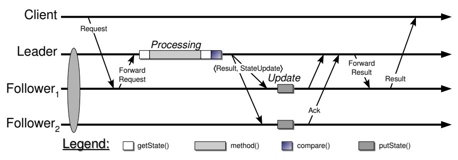
(a) With leader receiver



(b) With no state change



(c) With failover



(d) With follower receiver after a failover

Fig. 4. The Leadercast protocol

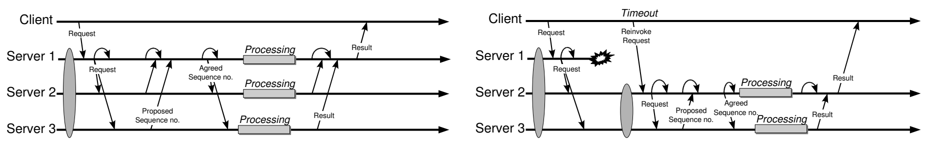
can happen if the leader failed before the followers could be informed about the original invocation. This forwarding to the current leader will only occur once per client, since the result message contains information about the new leader, and hence the client-side proxy can update its contact server.

As discussed above, the client-side proxy is responsible for selecting the contact server. For the leadercast protocol, the group leader (primary) is selected unless it has failed. The server selection strategy is embedded in the invocation semantic representation associated with each method. When the client detects that the leader has failed, the choice of contact server is random for the first invocation; the new leader is then obtained from the invocation reply and future invocations are directed to the current leader.

5 The Atomic Multicast Protocol

The atomic multicast protocol implemented in the context of this thesis is based on the ISIS total ordering protocol [4], hence only a brief description is provided herein. The protocol is useful to ensure that methods that modify the shared server state do so in a consistent manner. Methods using the atomic protocol must behave deterministically to ensure consistent behavior. The protocol is a *distributed agreement protocol* in which the group members collectively agree on the sequence in which to perform the invocations that are to be ordered. Fig. 5(a) shows the protocol. In the first step, the client sends the request to a contact server, who forwards the request to the group members, each of which respond with a *proposed sequence number*. The contact server selects the *agreed sequence number* from those proposed and notifies the group members; the highest proposed sequence number is selected. Finally, when receiving the agreed sequence number each member can perform the invocation and return the result(s) to the contact server, which will relay it to the client.

The contact server selection strategy is random for load balancing and fault tolerance purposes. The contact server acts as the entity that defines the ordering of messages, and serves this function for all invocations originated by clients using it as the contact



(a) The Atomic protocol (b) A failover scenario

Fig. 5. The Atomic multicast protocol

server. Since the choice of contact server is random, the same client may choose a different one for each invocation that it performs. It follows that also different clients will use different contact servers. An alternative contact server selection strategy is to always select the same server (the leader) to do the message ordering. By doing so, a *fixed sequencer* protocol requiring less communication steps can be implemented. The fixed sequencer and other total ordering protocols are discussed in [5].

Fig. 5(b) illustrates one scenario in which the contact server fails before completing the current ordering. The client detects the failure of the contact server, and sends the request to an alternative server. In this particular scenario, the remaining servers needs to rerun the agreement protocol. However, had the contact server failed after completing the agreement protocol, but before emitting the result to the client, the new server must emit the previous result in response to a reinvocation.

The two-step communication approach used for EGMI between the client and the group members precludes the provision of a true active replication scheme. In particular, the client-side proxy will not receive replies directly from all the servers, and thus cannot mask the failure of the contact server towards the client-side proxy. Hence, if the contact server fails during an invocation, the client-side proxy is required to randomly pick another server and perform a reinvocation. The failure of the contact server, however, is still masked from the client object. But the disadvantage is that the failover delay of the atomic approach is equivalent to that of the leadercast approach when the contact server fails. However, one way to provide true active replication is to let clients become (transient) members of the object group prior to invoking methods on it, allowing clients to receive replies from all members and not just the contact server. It is foreseen that the client-side proxy can hide the fact that it has joined the object group, from the client object before performing an invocation, e.g. by annotating the method with @Atomic(join=true). An optional leaveAfter attribute could also be provided indicating the number of invocations to be perform before the client-side proxy requests to leave the group. This way true active replication can be provided also to clients.

6 Runtime Adaptive Protocol Selection

Another useful mechanism that can easily be implemented in this architecture is support for *dynamic runtime protocol selection*. Dynamically changing the replication protocol of methods at runtime is useful for systems that wish to dynamically adapt to changes in the environment. For instance, a server may decide to change its replication protocol for certain methods to improve its response time, if the system load increases. One might also imagine a special module that can configure the replication protocols of a server group remotely from some management facility (e.g. ARM [11]) to adapt to changing requirements. For example, if moving to more powerful hardware, one can simply migrate replicas to the new hardware, followed by a change of the replication protocol to use for certain methods. This section briefly outlines how this feature can be implemented.

First, a @Dynamic marker is needed, which must be added to methods that should support dynamic reconfiguration. Next, the Dynamic replication protocol must be implemented, which is simply a wrapper for the other supported protocols. The Dynamic protocol must maintain a mapping for each @Dynamic method and its currently configured invocation semantic. By default, methods that declare @Dynamic should be configured with the @Anycast semantic, unless the marker is parametrized with the desired default protocol, e.g. @Dynamic(protocol=@Leadercast). A DynamicReplicationService interface can be provided that enables the server (or other protocol modules) to dynamically change the invocation semantics of the server's methods at runtime. A protocol module may then implement update algorithms that can seamlessly reconfigure the replication protocol of individual methods at runtime. One scheme could be to change the replication protocol of certain methods based on the size of the group. For example, if the group only has three members or less then @Atomic is used; if it has more than three members then @Leadercast is used.

Another, more subtle use of this feature relates to a client designed for testing the performance of various replication protocols. The server can then simply implement a set of test methods, each declaring the @Dynamic marker, whereas the client can invoke a special method to set the appropriate replication protocol to be tested, before invoking the actual test methods on the server. To allow clients to reconfigure the replication protocol of methods, the server (or a module) must provide a remote interface (e.g. by exporting the DynamicReplicationService interface) through which clients can update the invocation semantics of the server-side methods.

7 Conclusions

This paper presented an architecture and accompanying implementation of a dynamic protocol selection mechanism that makes it flexible and easy to improve the resource utilization of replicated services, by taking advantage of application semantics. The features of this architecture may also be used to support self-optimization by runtime reconfiguration of replication protocols for each individual server method.

References

1. Amir, Y., Danilov, C., Stanton, J.: A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In: Proc. Int. Conf. on Dependable Systems and Networks (June 2000)
2. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language, 4th edn. Addison-Wesley, Reading (2005)
3. Ban, B.: JavaGroups – Group Communication Patterns in Java. Technical report, Dept. of Computer Science, Cornell University (July 1998)

4. Birman, K.P., Joseph, T.A.: Exploiting Virtual Synchrony in Distibuted Systems. In: Proc. 11th ACM Symp. on Operating Systems Principles (1987)
5. Défago, X.: Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Number 2229 (August 2000)
6. Felber, P.: The CORBA Object Group Service: A Service Approach to Object Groups in CORBA. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland (January 1998)
7. Felber, P., Défago, X., Eugster, P., Schiper, A.: Replicating CORBA Objects: a Marriage Between Active and Passive Replication. In: Proc. 2nd Int. Conf. on Dist. Applic. and Interop. Systems (June 1999)
8. Felber, P., Jai, B., Smith, M., Rastogi, R.: Using semantic knowledge of distributed objects to increase reliability and availability. In: Proc. 6th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS) (January 2001)
9. Garcia-Molina, H.: Using semantic knowledge for transaction. Processing in a distributed database 8(2), 186–213 (1983)
10. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. IEEE Computer 30(4), 68–74 (1997)
11. Meling, H.: Adaptive Middleware Support and Autonomous Fault Treatment: Architectural Design, Prototyping and Experimental Evaluation. PhD thesis, Norwegian University of Science and Technology, Dept. of Telematics (May 2006)
12. Meling, H., Helvik, B.E.: Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model. In: Proc. 23rd Int. Performance, Computing, and Comm. Conf. (April 2004)
13. Montresor, A.: System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems. PhD thesis, Dept. of Computer Science, University of Bologna (February 2000)
14. Moser, L.E., Melliar-Smith, P.M., Narasimhan, P.: Consistent Object Replication in the Eternal System. Theory and Practice of Object Systems 4(2), 81–92 (1998)
15. Ren, Y., et al.: AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. IEEE Trans. Comput. 52(1), 31–50 (2003)
16. Sommers, F.: Call on extensible RMI: An introduction to JERI (December 2003), `http://www.javaworld.com/javaworld/jw-12-2003/jw-1219-jiniology_p.html`
17. Tanenbaum, A.S., van Steen, M.: Distributed Systems – Principles and Paradigms. Prentice Hall, Englewood Cliffs (2002)