Abstract

The Java Remote Method Invocation (JRMI) specification simplifies the development of distributed Java applications, but provides little support to guarantee reliable, highly-available operation. The Aroma System is middleware that transparently enhances the JRMI model with the mechanisms required for consistent replication of client and server objects. By exploiting novel interception mechanisms, the Aroma System is activated at runtime, with minimal modification to the application and to the JRMI infrastructure. The Aroma System adapts the JRMI model for group communication, and exploits an underlying reliable, totally-ordered multicast protocol to provide strong replica consistency. In this paper, we describe the architecture of the Aroma System, and discuss the mechanisms required to achieve transparent replication of Java RMI objects and to enforce strong replica consistency.

1. Introduction

The Java platform is ideal for developing distributed applications for heterogeneous platforms. The distributed Java object model (DJOM) [14] eases distributed application development by providing mechanisms for remote method invocation, object serialization and distributed garbage collection. However, the model does not define any mechanisms that can be exploited by the application for reliable, highly-available operation.

Object replication is a well-known effective approach for improving the fault-tolerance and availability of distributed applications. There has been extensive research on object replication in Java. However, most approaches either propose the addition of new semantics to the DJOM to support replication, or promote an object caching strategy to improve availability and performance. With the former approach, the developer must be highly conversant with the semantics of the new model, and must reprogram his applications to exploit the mechanisms correctly.

Achieving consistent behavior in a replicated system is not an easy task. Ideally, the mechanisms for replication should be integrated into the DJOM infrastructure, allowing distributed applications to take advantage of those mechanisms implicitly. The Aroma System adopts this approach, exploiting novel interception mechanisms to enhance the existing DJOM with support for replication at runtime. Because this approach does not modify the DJOM semantics, it can be exploited by both existing systems and new applications to obtain fault-tolerant, highly-available operation.

To provide fault tolerance, the Aroma replication mechanisms are enhanced to include support for fault detection and recovery from faults. This paper focuses on the design and implementation of the core Aroma replication mechanisms, and does not discuss Aroma’s support for recovery.

2. The Java RMI Architecture

The Aroma System targets distributed Java applications that conform to the Java Remote Method Invocation (JRMI) specification [13]. The JRMI model masks remote communication between Java Virtual Machines (JVMs), enabling clients in one JVM to invoke the methods of servers in a different JVM exactly as though they were local. The current release (1.3) of the Java2 platform provides two distinct implementations of the JRMI specification that differ in their choice of transport protocol. The RMI-JRMP implementation uses the Java Remote Method Protocol (JRMP). JRMP exploits Java-specific functionality such as distributed garbage collection and, therefore, is suitable for pure Java client-server applications. On the other hand, the RMI-IIOP exploits the CORBA Internet Inter-ORB (IIOP) [11] protocol, allowing RMI-IIOP objects to interoperate with CORBA2.3-compliant applications.

The generic JRMI architecture is shown in Figure 1. The proxy layer consists of a server-side skeleton and a client-side stub, and provides the interface to the appli-
Figure 1. The Java Remote Method Invocation Architecture.

cation. The remote reference layer resolves object references, and validates the protocol-specific semantics, and the transport layer establishes and maintains the physical TCP/IP connections between client and server objects.

A JRMI remote object implements one or more Remote interfaces that, collectively, define the methods that can be invoked on that object by a remote client. The remote object is accessed using a remote reference that consists of a TCP/IP endpoint, and an “object identifier” unique to the object’s JVM. A stub implements the same set of remote interfaces as the server, and caches the server reference. Clients retrieve and install the stub, using it as a server proxy on which they make invocations, locally. The stub transparently sets up the TCP/IP connection to the server endpoint, and forwards the invocation to the server-side skeleton. The skeleton, acting as a client proxy, invokes the method locally on the server and marshals the results back to the stub, which uploads them to the client. Thus, the proxy layer masks all remote communication from the application.

The JRMI model also exploits a nameserver for bootstrapping purposes. The nameserver – represented by the rmiregistry in RMI-JRMP and by the nameserv in RMI-IIOP – maintains a transient database of name-stub mappings for local servers. Clients, equipped with the server name, lookup the nameserver to retrieve the corresponding stub. Once bootstrapping concludes, the nameserver plays no further role in client-server communication.

3. Replication of JRMI Objects

Replication involves distributing multiple copies, or replicas, of the application objects over different Java Virtual Machines (JVM) on potentially different processors. There are different approaches to replication, based on the level at which the mechanisms are implemented. A service approach implements the replication mechanisms as packages at the application level; the application developer must be familiar with the replication model, and is required to write code on a per-application basis to exploit these services. The integration approach pushes the replication mechanisms into the JRMI infrastructure. Application developers can exploit the infrastructure for fault tolerance; however, this approach typically requires modification of the JRMI infrastructure. A third approach, used by the Aroma System, adopts the use of interception to capture and reroute the JRMI messages exchanged between the objects of the application and the network, in a transparent manner. By exploiting interception, Aroma enhances the JRMI model with support for replication, with little modification of either the application or the JRMI infrastructure. The interception approach is described in detail in Section 4.

3.1. Replication Entity

The Aroma middleware intercepts all remote method invocations, and exploits a reliable, totally-ordered multicast protocol (see Section 6) to forward these invocations to all replicas of the remote object. Because all replicas see the same invocation messages in the same order, remote method invocations that alter the state of the server will do so in a consistent manner across all replicas of the server. However, the Aroma System cannot intercept invocations that are local to the JVM, or detect local operations that directly access member variables.

To ensure that such local accesses do not result in inconsistent state across replicas, we employ a replication “entity” as the basic unit of replication. A replication entity is governed by the following rules.

- *An entity consists of a collection of tightly coupled objects hosted in the same JVM.* Thus, the smallest possible entity, a singleton, will contain a single object.
- *An entity must contain exactly one JRMI server.* Thus, if a JVM hosts two server objects, there are two replication entities for that JVM.
- *The JRMI server defines the single point of entry to an entity.* The only external interactions with the entity consist of remote method invocations on the JRMI server.
4. The Aroma System

Figure 2 shows an overview of the Aroma System highlighting its four main components, namely, the Aroma Interceptor, the Aroma Parser, the Aroma Message Handler and the Aroma Multiplexer.

- **Internal access.** Given an entity consisting of two objects A and B, B can make local method invocations on A, or can modify the state of A through direct access to its member variables. The internal access is contingent on B satisfying the access modifiers (public, protected, package) for A.

- **All method invocations and member modifier operations within the entity must occur deterministically.** If all replicas of an entity start from a consistent state, and see the same sequence of externally-generated invocations, they must each respond by executing the same sequence of internal operations. These internal operations must produce identical results at all replicas of the entity. Thus, once the invocation has been handled successfully, all replicas of the entity must have transitioned to a consistent final state.

- **The internal state of the entity is the collective internal states of its member objects.**

In the following sections, the term “object” is synonymous with “entity,” unless explicitly specified otherwise.

- **The Aroma Interceptor** is a software component embedded within the transport layer of the JavaRMI protocol stack, and activated at runtime. The Interceptor captures all TCP/IP calls originating from the distributed Java application, including the socket lifecycle calls (i.e., socket creation, bind, listen, connect, accept and close) as well as the socket I/O calls (i.e., read and write) that involve actual JRMI data messages. The Aroma Interceptor component is the key to deploying the Aroma System in a manner that guarantees transparency to the application.

  The Interceptor interacts with an internal Delegate object that wraps the standard implementation of the intercepted calls. A derived AromaDelegate object overrides select wrapper methods to divert the corresponding calls to the Aroma mechanisms represented by the Aroma Message Handler component in the figure. Thus, while the Delegate mode provides normal unreplicated JRMI operation, the AromaDelegate mode effectively triggers the Aroma mechanisms. The details of the Interceptor and the associated overheads can be found in [9].

- **The Aroma Parser** is a simple message-parsing component that is equipped with the formats used by the JRMP and the IIOP protocols. The Parser component is exploited in two different modes at different levels of the Aroma architecture.

  In conjunction with the Interceptor, the Parser is used in read-only mode to analyze intercepted TCP/IP messages. Because the Interceptor captures ALL TCP/IP calls, the Parser component is exploited to selectively intercept only JRMI-specific calls, while allowing all other TCP/IP calls to continue along their default TCP/IP communication paths.

  In conjunction with the Message Handler, an enhanced Parser component is exploited in read-write mode to “patch” outgoing messages and to undo the patching on incoming messages. Patching messages is a complex process that involves selectively modifying the contents of intercepted messages. As ex-
explained in Section 8.3, this mechanism is required in Aroma to enforce replica consistency despite the existence of sources of non-determinism present in the JRMI model.

- The Aroma Message Handler implements most of the mechanisms required to provide consistent replication for JRMI application, and is the core component of the Aroma System. Its functionality includes:
  - Maintaining the mapping between replica-specific identifiers and their associated group identifiers.
  - Adapting the intercepted point-to-point JRMI messages for one-to-many transmission, over a reliable, totally-ordered multicast protocol, namely Totem [7].
  - Implementing scheduling and dispatching mechanisms to automate routing of messages from the multiple JRMI objects onto the single multicast protocol interface. This activity is coordinated with the Aroma Multiplexer mechanisms.
  - Implementing mechanisms to enforce replica consistency.

- The Aroma Multiplexer provides the interface to Totem, and is exploited by the Message Handler to map messages from multiple JRMI servers, onto the multicast protocol interface.

In the following sections, we elaborate on some of the mechanisms implemented by the Message Handler component, to enforce replica consistency.

4.1. Replication Mechanisms

The principal objectives of the Aroma System are transparency and strong replica consistency. A third issue is replication style. The choice of replication style is governed by the availability of resources and the tolerable latency of recovery from failure. The Aroma System supports both active and passive replication, as described in Section 4.3.

In the context of replication, transparency ensures that the replication mechanisms are completely hidden from the application. Thus, at the application level, every replica behaves exactly as though it were an unreplicated JRMI object using standard JRMI semantics. In the Aroma System, we achieve this transparency by exploiting interception, and by adopting the object group paradigm as discussed in Section 4.2. Replica consistency mechanisms ensure that all replicas of a JRMI object maintain the same state at all times. The Aroma System exploits a reliable, totally-ordered multicast protocol to facilitate replica consistency, as described in Section 6.

4.2. Object Groups

An object group is an abstraction for a collection of objects that implement a common interface, and thus provide some common functionality. The Aroma System exploits the object group pattern [5] for the transparent replication of JRMI objects. The replicas of an object correspond to the members of the object group; the group interface is provided by the collective remote interfaces implemented by the JRMI object. Although object groups can be heterogeneous (i.e. contain members that implement different sets of interfaces), for the remainder of this paper the term “object group” is synonymous with a homogeneous object group, and represents a replicated object. The object group interface conceals the details of the replication – such as the location and distribution of replicas, and the replication style – from both the clients and the replicas themselves.

Typically, replicas are instantiated on distinct processors. They have different object identifiers and endpoints, and hence different object references. To reconcile these differences to produce a unified group reference, the Aroma Message Handler generates a unique group identifier for every replicated object; it then maintains, locally at each replica, the correspondence between this group identifier and the replica’s object reference. The Aroma Message Handler contains a handle to a “group manager” that administers object groups, detecting the presence of new replicas, and ensuring that the states of replicas within a group remain consistent. The group manager mechanisms are responsible for detecting the failure of a replica, for triggering corrective action that includes initiating recovery.

4.3. Replication Style

The choice of replication style (active or passive) is influenced by the tradeoff between resource consumption and the latency associated with recovery from faults.

- Passive Replication

In passive replication, a single primary replica executes all invocations, with one or more secondary replicas poised to take over from the primary, if the primary fails. Every invocation (response) is multicast to all replicas of the passively replicated server (client). Periodically, the state of the primary is retrieved and multicast to the group, and logged locally at each replica, by the Aroma infrastructure.
The checkpoint also triggers the garbage collection of all invocations and responses logged prior to the checkpoint, effectively pruning the logs. Recovery involves the election of a new primary, its initialization to the last checkpoint of the primary, and the subsequent replay of logged messages to restore its state to the point just prior to the failure.

- **Active Replication**
  In active replication, all of the replicas of an object are loaded into memory, and operate concurrently. Invocations and responses are multicast to all of the replicas; every server replica executes every invocation, and returns a response. The client accepts the first response that it receives, and discards the remaining identical responses. Thus, the occurrence of a fault at one replica is masked by the continued operation of other replicas. Active replication provides the fastest recovery from faults; the observed latency of an operation is equal to the execution time of the fastest live replica, despite the occurrence of faults in the system. Active replication is desirable in real-time or mission-critical applications. Typical applications can involve upwards of 5000 objects; in this respect, active replication is expensive, being resource-intensive in terms of memory consumption and CPU utilization.

5. **Replica Consistency**

The implicit assumption behind replication is that all replicas of an object are indistinguishable in functionality and behavior. All of the replicas must maintain consistent state, and must handle invocations in an identical manner, returning consistent responses. Replica consistency mechanisms are required for both passive and active replication. We define the following rules for replica consistency:

- **Replicas must behave in a deterministic manner at all times.**
  Two replicas of a deterministic object that start from a consistent initial state, and execute the identical sequence of invocations, are guaranteed to attain a consistent final state. Non-determinism is caused by multithreading, or by the use of replica-specific information such as system time or hostname. If sources of non-determinism exist, they must be sanitized to give the effect of non-deterministic behavior.

- **Replicas must start operation from a consistent initial state.**
  The definition of “consistent” state depends on the context of its usage. In normal replication, all replicas must have identical initial state at system startup. However, during recovery from a failure, a new or recovering replica will start from the last checkpointed state, and subsequently replay logged invocations and responses to restore its state to the point where it can process new messages.

- **Replicas must execute the same sequence of invocations.**
  By stipulating that all replicas start from a consistent initial state, as well as by subsequently ensuring that they see the same sequence of invocations, we guarantee that all updates to internal state occur in a consistent manner across all replicas.

- **Replicas must process each distinct invocation only once.**
  In active replication, every server replica executes every invocation and returns a response, resulting in duplicate responses at the client. In passive replication a recovering primary server replica might receive a repeat invocation of an invocation already processed by the failed primary. Such duplicate messages must be detected and discarded; failure to do results in multiple executions of the same invocation, causing incorrect behavior and possible corruption of the internal state of the replicated object.

The Aroma System strictly enforces these rules, exploiting mechanisms that ensure deterministic behavior, state transfer mechanisms to guarantee consistent initial state, logging mechanisms to ensure consistent state updates during recovery, a reliable totally-ordered multicast protocol to ensure that all replicas “see” the same sequence of invocations and responses, and finally, duplicate detection mechanisms that suppress duplicate messages. In the following sections, we discuss the implementation of these mechanisms in more detail.

6. **Reliable Totally-Ordered Multicast**

Standard JRMI implementations employ, by default, a point-to-point (TCP/IP) transport for client-server communication. Replication necessarily involves one-to-many communication (i.e., multicast) from one client object to the multiple replicas of the server object. To achieve replica consistency, all replicas (members) of a replicated object (object group) must “see” the same sequence of messages in the same order; thus, they will perform the same operations, resulting in the same state being maintained across all of the replicas.

One way of achieving the effect of multicast is to use multiple TCP/IP connections from the client object,
with as many connections as there are server replicas. The client object would establish and maintain a separate TCP/IP connection to every server replica for the entire duration of the client-server communication. Furthermore, for replica consistency, ordering and consensus algorithms will be required over and above the TCP/IP communication. Clearly, this solution is impractical in terms of scalability, performance, and the effort expended in connection management, ordering and consensus.

Instead, the Aroma system exploits the services of a reliable, totally-ordered multicast group communication system, in our case, Totem [7], for communication between and within object groups. Totem’s reliable delivery guarantees that every message is received by all of the replicas. Totem’s total ordering guarantees that messages are delivered in the same sequence to all replicas. Totem also implements a membership protocol that detects faults at the processor level and reconfigures the system.

The mapping of TCP/IP calls onto the Totem interface is achieved in a transparent manner by exploiting the Aroma Interceptor. The Interceptor’s Delegate object defines the standard TCP/IP socket interface exploited by the JRMI transport, as shown in Figure 3. The derived AromaDelegate object overrides select methods of the Delegate superclass to map the TCP/IP calls onto the “connection interface” to the Aroma Message Handler; at present, we implement this interface using Unix sockets. The Aroma Message Handler subsequently adapts these messages for transmission over the Totem group communication system.

### 7. Duplicate Detection

Duplicate detection mechanisms detect, and suppress, duplicate messages (invocations and responses) at the source, and failing that, at the destination. Duplicate detection is exploited both in actively and passively replicated systems, under different scenarios.

- An actively replicated system requires duplicate detection for normal operation. Every replica of an actively replicated server executes every invocation, and returns a response. Thus an actively replicated client will generate duplicate invocations, and an actively replicated server will generate duplicate responses.

- A passively replicated server requires duplicate detection under recovery conditions. Consider the case when a primary server replica processes an invocation, and fails prior to returning a response. The lack of a response could prompt a re-invocation, from the client to the newly recovered primary, for a request that has already been handled.

In both cases, failure to detect and discard duplicate messages results in repeated processing of the same operation, leading to incorrect system behavior and possible corruption of state. To facilitate the detection and suppression of duplicate messages, the Aroma Message Handler appends “operation identifiers” to every outgoing message. These operation identifiers must be generated in a deterministic manner, and must be globally unique to ensure detection. It is also desirable to establish bounds on
the size of operation identifiers for simplicity and performance. Operation identifiers are generated for invocation messages, and subsequently carried by both the invocation and the resulting response.

The format of the operation identifier is shown in Figure 4, and consists of a timestamp and a sequence number. The timestamp is generated by the underlying multicast protocol, and pertains to the timestamp of the parent message that triggered the invocation. The timestamp is globally unique and is itself, a deterministic value that can serve to identify the invocation. However, a single parent message could spawn multiple child operations, each of which is now associated with the same timestamp. Thus, the Aroma mechanisms generate a “sequence number” that identifies the sequence in which the child operations are initiated, for a given parent message. The combination of timestamp and sequence number uniquely identifies the invocation message. Because there is a one-to-one correspondence between invocations and responses, responses simply exploit the operation identifiers of their corresponding invocations as shown in Figure 4.

8. Sources of Non-determinism in Java RMI

A non-deterministic operation is one whose result cannot be reproduced by independent executions of the operation. In the context of replication, this implies that a non-deterministic operation could generate different results at different replicas of the same object. For replica consistency using Aroma, we require deterministic operations within each replicated entity. Given this deterministic behavior of entities, by extension, the content of invocations and responses generated by the entities will also be deterministic. Therefore, non-determinism – if it exists – must be due to the actions of the JRMI infrastructure. By studying the communication formats in an unreplicated JRMI system, we identify some sources of non-determinism and propose mechanisms to address them.

8.1. Object Identifiers

An object identifier pinpoints the object’s location within a JVM on a specific host. The formats of the ObjID and the ObjectKey, object identifiers specific to RMI-JRMP and RMI-IIOP, respectively, are shown in Figure 5.

The “Unique Identifier” (UID) of the ObjID, and the “ServerID” component of the ObjectKey, server to identify a JVM uniquely, on a given host. These identifiers are generated using current system time as an input, to ensure uniqueness. Because the replicas are distributed across different processors, this JVM identifier is not guaranteed to be consistent across these processors. The “object number” component of the ObjID, and the “User Key” component of the ObjectKey, pinpoint a specific object within the target JVM. These values are determined by the order in which servers are started within the JVM. Because this order is application-specified, the value is deterministically assigned at all JVMs. However, the existence of the UID/ServerID component contributes to non-deterministic behavior.

As shown, object identifiers are embedded in every invocation header, and validated at the destination before the JVM forwards the request to the specified server. Altering the object identifier results in the JVM invalidating and discarding the request. However, retaining the default identifiers poses a problem; because the invocation header can carry only one object identifier, this request...
can be handled by only the single replica represented by this identifier. The request would be invalidated at the JVMs hosting the other replicas, leading to inconsistent state across all of the replicas.

8.2. Server References

The server reference – represented by a LiveRef in RMI-JRMP and by an IOR (Interoperable Object Reference) in RMI-IIOP – encapsulates the object identifier and a TCP/IP endpoint for the server. Just as the object identifier is inherently replica-specific, the TCP/IP endpoint is necessarily different across different replicas because they are hosted on distinct machines. A client exploits this reference to establish a connection to the server, and subsequently, to dispatch invocations to it. With different replicas establishing different references (based on their TCP/IP endpoint alone), we require some mechanism to reconcile the replica references into a single “group reference.” At each replica, the inverse mapping – from the group reference to the replica reference – must be provided. The client would be allowed to access, and use, only the group reference.

8.3. Enforcing Determinism Using Aroma

At present, the Aroma System enforces determinism in these cases by “patching” JRMI messages. Every replica-specific identifier is mapped to a group-identifier that is generated in a deterministic manner across all replicas. The Aroma infrastructure at each processor maintains the group-to-replica identifier mappings for all local replicas on that host. The Aroma Patcher, derived from the Aroma Parser component, is a component of the Message Handler that screens all incoming and outgoing network messages. Replica identifiers on outgoing messages are replaced by the group equivalents; this mapping is reversed for incoming messages.

In a typical scenario, an unreplicated client invokes an unreplicated server using a previously retrieved server reference. With the Aroma infrastructure in place, the mechanisms at the client-side Message Handler determine the corresponding group reference, patch the message header accordingly and multicast the message to all replicas of the object. The Aroma mechanisms at the server-side Message Handler map the group identifier back to the identifier associated with the local replica, and forward the invocation to it. Thus, every Message Handler knows only the local replica reference and the group reference to which it maps. Further, the patching is completely transparent to both the client and the server.

Patching is a complex, inelegant and relatively expensive mechanism. Ideally, both the simplicity and performance of replicated applications would be improved if hooks – in the form of externally-triggered mechanisms, or explicit API-level support – could be provided, by the JRMI implementors, to enforce the deterministic assignment of these identifiers.

9. Additional Challenges

The issues that we have addressed so far have focussed on the replication mechanisms, and on achieving replica consistency. However, the semantics of the JRMI model pose additional challenges.
9.1. Unreliable Nameserver

The nameserver expedites the process of setting up the client-server communication and, as a result, constitutes a single point of failure. Failure of the nameserver renders new clients incapable of retrieving the references they require, to contact fully operational servers. Previous research [1, 6] proposed explicit mechanisms to implement a fault-tolerant nameserver. However, from Aroma’s point of view, the nameserver is just another JRMI object, with a standard well-known object identifier. Therefore, we can directly exploit the Aroma mechanisms to provide a replicated – and, therefore, fault tolerant – nameserver.

9.2. Distributed Garbage Collector

At present, distributed garbage collection is supported only by the RMI-JRMP model. Every JVM that hosts at least one remote server also hosts a distributed garbage collector (DGC) with a standard object identifier. The DGC operates as follows:

- Client-side JRMI mechanisms make a dirty() call on the DGC upon retrieving a server reference from that JVM. The DGC returns a lease, and increments the reference count for that server.
- Client-side JRMI mechanisms periodically renew the lease, and make a clean() call on the DGC when there are no more clients using that reference.
- The DGC responds to a lease timeout, or to a dirty() call by decrementing the reference count for the server. If the server reference count is zero, the server is eligible for garbage collection.

The DGC associates every client with a Virtual Machine Identifier (VMID) which, similar to the Unique Identifiers used by the ORB, uses a time-based value to provide a unique identifier to each JVM. Thus, VMIDs are non-deterministic in nature. Furthermore, the lease-renewal is controlled by timeouts. Different client replicas could generate lease-renewal calls for the same server, at different times. Because these calls are interspersed between regular invocations, our duplicate detection mechanisms cannot use the operation identifiers to differentiate between a duplicate call and a new lease-renewal call. As a preliminary solution, we propose to patch VMIDs to enforce determinism, and to parse lease-renewal messages to enforce stricter duplicate detection and suppression at the client JVM. A side-effect of this strategy is that it requires the Aroma Parser to understand the semantics of the object serialization protocol, used by RMI-JRMP to implement pass-by-value semantics for exchanging parameters and results.

9.3. Multithreading

The RMI-JRMP model makes extensive use of multithreading, on the server-side, to service multiple client requests concurrently. The specification offers no guarantees on the scheduling of invocations at the server. Thus, two concurrent requests from the same client JVM could be scheduled sequentially (on one thread), or could be scheduled on separate threads and serviced in a non-deterministic order. Two concurrent requests from different client JVMs will be scheduled on different threads, but their order of execution is not deterministic. If the invocation involves a change in internal state at the server, then the order of invocations must be preserved at all replicas, to maintain replica consistency. Because we intercept all invocations destined for a server, we can impose a strictly unithreaded model by scheduling only one invocation at a time, per server. At present, we do not handle nested invocations and callbacks that involve multithreading.

10. Related Work

The Aroma System builds on the experience gained in the development of the Eternal System [8, 10], but is distinct from it in that Eternal provides fault tolerance for CORBA applications, while Aroma provides fault tolerance for Java RMI applications. The Eternal System is the first concrete implementation of the OMG specification for Fault Tolerant CORBA [12]. It provides transparent, consistent replication of CORBA objects, without modifying either the application or the ORB.

Current research work for Java adopts different approaches to object replication. Most approaches advocate new semantics or the addition of keywords or class- es to the distributed Java model to facilitate replication. Goldrush [2], which is aimed at mobile applications, exploits replication for availability. Clients cache replicas of database entities hosted at a central server. When disconnected from the server, all transactions are made on the local copies; upon reconnection, the logged transactions are replayed at the server and conflicts are reported to the client for resolution.

An alternative approach to transparent replication advocates the use of reflection and metaprogramming. The FRIENDS architecture [3] provides a library of metaclass classes that can be exploited to provide fault tolerance, secure communication and group distribution management services to an application. The Metajava project [4] applies the metaprogramming approach to the Java model and describes algorithms that can be exploited to achieve both active and passive replication of Java objects.

Two systems that have similar objectives to the Aroma System are Filterfresh and JGroup. Filterfresh [1]
promotes replication of stateless JavaRMI server objects by exploiting the services of a fault-tolerant registry (FTRegistry) and a smart stub object. By adding new classes to the Java model, Filterfresh provides the ability for multiple server replica references to be bound to a single name in the FTRegistry. Upon detecting a server fault, the smart stub which is downloaded to the client looks up an alternative server reference for the server group, and re-initiates the invocation. Thus, server faults are transparent to the client.

JGroup [6] also exploits the object group paradigm, introducing a ReplicatedRemoteObject class that should be subclassed to take advantage of the replication capabilities. A GroupRef holds references to all replicas in that group, and implements reliable unicast semantics to provide failover. Clients make invocations on the GroupRef, resulting in the method being invoked on one of the replica references. If an exception occurs, the method is invoked sequentially on the remaining references until it succeeds. Both JGroup and Filterfresh require new semantics to be used by the application to exploit replication.

The Aroma System is the only solution that provides transparent replication of Java RMI objects – requiring little modification to either the application or the JRMI model – and that enforces strong replica consistency.

11. Conclusion

The Aroma System is middleware that enhances the Java Distributed Object Model with support for object replication. The Aroma System makes three significant contributions to research in replication for Java RMI objects:

- Aroma exploits a novel interception approach to provide transparent replication of JRMI objects.
- Aroma extends the JRMI model with support for object groups, and adapts JRMI messages for reliable totally-ordered multicast.
- Aroma enforces strong replica consistency, implementing mechanisms to overcome sources of nondeterminism specific to the JRMI model.

The Aroma System can be deployed at runtime with little modification to the application or to the JRMI infrastructure. Therefore, both existing distributed Java systems and new JRMI applications can exploit the Aroma mechanisms for fault tolerance and highly available operation.

References


