A Type System Extension for Middleware Interactions

Sven De Labey and Eric Steegmans
K.U.Leuven, Dept. of Computer Science
200A, Celestijnenlaan, B-3000 Leuven, Belgium
{svendl,eric}@cs.kuleuven.be

ABSTRACT
Object-oriented programming languages such as Java provides inadequate support for advanced method invocation strategies in distributed applications. Invocation semantics such as reliable unicast and multicast must be implemented based on primitive, unreliable unicast mechanisms such as Java RMI and Socket communication. This forces developers to devise ad hoc communication strategies, which is a repetitive and error-prone process. Moreover, these communication strategies are entangled with the business logic of the application, making the code hard to read and maintain.

In this paper, we propose an extension of the Java type system based on communication qualifiers. These qualifiers decorate variables with information about the invocation semantics that must be used when methods are invoked on that variable. We show how communication qualifiers are used to support transparent multicast and reliable unicast invocation semantics in Java. Moreover, we introduce declarative operations to enable developers to fine-tune these communication strategies.

Categories and Subject Descriptors
D.1.5 [Programming Techniques]: Object-Oriented Programming; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Type system, Type qualifier, Fault tolerance, Group method invocation

1. INTRODUCTION
The introduction of the Java API for Remote Method Invocation [1] is an attempt to bring the semantics of local method invocation to the world of distributed systems. Local method invocation relies on an implicit unicast communication strategy. The strategy is implicit because the invocation semantics cannot be altered by the programmer.

It is based on unicast because each method invocation \( o \cdot m() \) is sent to exactly one receiver, namely the instance to which \( o \) refers. Although these semantics work fine for applications running on a local virtual machine, they become less appealing in a server architecture with replicated services\(^1\), where advanced invocation strategies like reliable unicast and multicast are frequently needed.

Reliable unicast is typically used to handle availability problems at the server side. Operations invoked on an unreachable service are reinvoked on one of the backup services until the request has been handled, or until the architecture runs out of backup services. In other words, reliable unicast semantics require the client to invoke an operation on any target from a group of compatible services. By relying on a single receiver invocation model, however, Java is unable to transparently support this feature, forcing developers to write code for explicitly rewiring the variable to a backup service and reinvoking the operation. The same problem occurs for multicast invocations, where an invocation must be sent to a all members of a group of services in stead of a single target.

In this paper, we present an approach based on communication qualifiers for altering method invocation semantics. Such a qualifier decorates a Java type and informs the middleware about the communication strategy that must be used when a method is invoked on that variable. Developers can fine-tune this strategy by using declarative operations, for instance, to enforce Quality of Service constraints on the members of a replicated service group.

This paper is structured as follows. In Section 2, we enumerate a number of shortcomings related to remote method invocation in Java. In Section 3, we discuss basic properties of communication qualifiers. Section 4 shows how these qualifiers are used to support multicast semantics, whereas Section 5 discusses how they enable reliable unicast. Section 6 shows how the behaviour of the communication mechanism can be altered at runtime. Section 7 briefly discusses related work and Section 8 concludes.

2. MOTIVATION
Without support for multicast and reliable unicast, a programming language forces programmers to reify low-level details of the server architecture in their source code. Indeed, the semantics of these communication strategies rely on the existence of multiple (replicated) service endpoints.

\(^1\)In this paper, we use the term services to refer to remote objects, web services, and any other invokable endpoint exported by a remote server.
and references pointing to these endpoints must be explicitiated in the source code before they can serve as backup invocation targets. This approach has a number of disadvantages:

- **Abstraction.** Algorithms supporting invocation semantics such as reliable unicast and multicast cooperate tightly with the business logic, thus obstructing a clean generalization into reusable libraries or aspects. Consequently, programmers have to implement ad hoc algorithms for supporting new communication strategies over and over again, leading to a very low level of abstraction.

- **Program Comprehension.** The code of these ad hoc communication protocols obfuscates the business logic and decreases the modularity of the source code, which in turn increases maintenance costs as well as the probability to introduce bugs in newer versions of the software.

- **Flexibility.** References pointing to replicas of a remote service must be managed in the source code, which increases the number of hardcoded references to physical service locations. This causes problems when services are migrated to other servers, it disables the discovery of newly added replicas, and it introduces problems when the application is reused in another server architecture.

The approach proposed in this paper advocates a different communication model, which is aimed at realizing three objectives related to object-oriented software development:

- **OO1 – Abstraction.** The plurality of replicated services must be hidden for application developers. Instead, they should invoke operations on a logical target without having to manage the underlying group of physical targets. Also, the algorithms that implement the required communication strategy must be hidden.

- **OO2 – Configurability.** Even though the details of a communication strategy must be hidden to increase the level of abstraction, the behaviour of those algorithms must be configurable by developers in order to customize the semantics of method invocation to the specific needs of an application.

- **OO3 – Compile-time Analysis.** A compiler must be able to analyze the flow of a communication strategy throughout the application in order to avoid communication conflicts. An example of such a conflict is an assignment of a unicast reference to a variable that requires a multicast method invocation protocol.

3. COMMUNICATION QUALIFIERS

In addition to the hierarchical class structure of an application, we introduce an independent hierarchy of communication qualifiers. These qualifiers decorate an instance variable with information on how the variable must behave as a method invocation target. A qualified variable var with type T and qualifier q is declared as q \( T \) var, where q represents one of the following communication strategies:

- **Multicast.** We introduce a communication qualifier, group, for stating that a call like target.m() must be sent to all services conforming to the group variable target. Such a group variable is declared as group target, and its semantics are discussed in Section 4.

- **Reliable Unicast.** We introduce two qualifiers, pool and sequence, to indicate that the call target.m() must be sent to any service conforming to the type of the pool variable until an instance successfully handles the request, or until each backup endpoint has been contacted without success. Such a pool variable is declared as pool \( T \) target or sequence \( T \) target and its semantics are discussed in Section 5.

The main difference between variables decorated by a communication qualifier and normal Java variables is that qualified variables are managed by the middleware. They are never explicitly initialized by developers. Instead, our compiler extension creates a stub for each qualified variable, similar to stub creation for remote interfaces in Java RMI [1]. All methods invoked on this stub are delegated to the middleware, which communicates with remote services based on the proper communication strategy. This strategy is composed by inspecting the communication qualifier and the type of the variable, as explained above (see also Figure 1).

Looking back to our design goals, communication qualifiers increase the level of abstraction (OO1) by shielding
developers from (1) the management of the underlying set of replicated services and (2) the technical details of the communication strategy. Moreover, communication qualifiers define a relationship between variables and their invocation semantics, thus allowing a compiler to analyze the flow of a communication strategy throughout an application (OO3).

Such an analysis enables the compiler to guarantee that unicast variables and multicast variables are never assigned to each other\(^2\), thus allowing for a clean separation between different method invocation semantics.

The remainder of this paper focuses on the definition of our communication qualifiers, and on their interactions with the middleware.

4. GROUP METHOD INVOCATION

We introduce a communication qualifier, group, to support multicast invocation semantics in Java. Group variables are declared by qualifying a variable with the group qualifier, for example group G groupVar. A method invoked on groupVar is transparently multicast to all services conforming to G (i.e. the type of the group variable).

**Group Filters.** In distributed software architectures, it often occurs that developers need to constrain group membership to those services that satisfy a list of functional requirements and/or Quality of Service constraints. Functional requirements relate directly to the business logic of an application, whereas QoS constraints relate to service availability, performance, and security. Such requirements can be enforced with group filters. A group filter is specified using a declarative operation, where, that accepts the filter as a boolean expression. During the construction of its communication strategy, the middleware filters the underlying collection of services and retains only those services that satisfy the filter condition. The resulting collection of services is used as a target for multicasting method invocations.

**Example.** Figure 2 shows an example of a group variable g of type S, which is used to multicast an operation f() to those services that conform to S. In order to qualify as an invocation target, a service must satisfy the membership condition, constr. At the code level, developers use the group qualifier to enable multicasting, and they use the where clause to specify membership constraints (constr).

The technical realization of this selective multicast semantics, however, is the responsibility of the middleware, which executes a communication strategy as follows. First, the middleware interacts with the registry in order to retrieve the group of services conforming to the type of the group variable, S (this is not shown in Figure 2). This group is filtered using the membership condition supplied by the developers (constr), leading to the removal of S1 and S3. Next, f is sent to S2 and S4 and the results of these method calls are collected into a new group variable p of type P. From now, operations invoked on p are sent to P1 and P2.

**Evaluation.** The introduction of the group qualifier has three advantages. First, it raises the level of abstraction (OO1) because developers are shielded from the details of implementing group management and multicast algorithms. This is done by substituting hardwired remote references with a stub. This stub shields developers from dealing with object migration and the discovery of newly replicated service instances. Second, programmers are able to fine-tune the multicast invocation semantics (OO2) by introducing membership constraints using the where clause. Third, by extending the type hierarchy, we enable the compiler to differentiate between unicast references (assigned to normal java variables) and multicast references (assigned to group variables). This enables static type checks on the communication infrastructure by performing a flow analysis on the communication qualifiers (OO3), thus avoiding unintentional assignments of multicast references to unicast variables and vice versa.

5. TRANSPARENT FAILOVER

The group qualifier signals the middleware to use multicast semantics for all methods invoked on the group stub. But replicated groups of services are also useful for supporting a reliable unicast invocation strategy. These semantics dictate that operations invoked on overloaded or unreachable services must be reinvoked on a backup service. Obviously, these semantics are incompatible with the semantics offered by the group qualifier, as the latter would contact all replicas at once. Therefore, we introduce a second communication qualifier, pool, to contact a group of interchangeable services one-by-one until a service handles the request.

5.1 Basic Service Pools

Variables with reliable unicast invocation semantics use the pool qualifier to notify the middleware that transparent fail-over is required for all methods invoked on that variable. Basic pool variables impose no order on the members of a service pool, so the middleware is allowed to execute a non-deterministic service selection algorithm. If a method invocation fails, the middleware randomly selects another service from the pool and transparently invokes the operation on that new target. Such a non-deterministic selection strategy is particularly useful in distributed architectures with actively replicated services because it enables both failover and load balancing among interchangeable services.

**Constraining Pools.** Pools and groups differentiate on the communication strategy they provide, but they use the same underlying collection of services. Therefore, we can...
reuse the `where` operation from Section 4 to impose membership conditions on service pools. A pool of Flight instances, for example, can be constrained to contain only those services that offer a flight from London to New York:

```java
pool Flight fs where fs.fliesRoute("LHR","NY");
```

### 5.2 Sorted Pools

In architectures where compatible services each have their own characteristics, developers may want to select a service based on service-specific properties. In that case, the `pool` qualifier is no longer appropriate because that qualifier assumes that all members have the same, replicated characteristics, which allows for randomized service selection. Therefore, we provide a second qualifier, `sequence`, to indicate that the middleware must use a deterministic service selection algorithm. This selection is driven by a list of quality attributes and the service that best approximates them is selected as the primary target for method invocation. Developers specify these quality attributes using a declarative operation, `orderby`, which accepts a Java primitive type, or a `Comparable` expression. For example, selecting the `Flight` service that offers the cheapest flight from London to New York, is implemented as follows:

```java
sequence Flight fs where fs.fliesRoute("LHR","NY")
orderby fs.getPrice();
fs.bookFlight();
```

Figure 3: Declarative language constructs guide the middleware during pool filtering, optimal service selection, and fault-tolerant method invocation.

**Example.** Figure 3 gives an example of middleware support for imposing constraints on service pools and for specifying a deterministic service selection policy. It shows how an instance variable `p` of type `S` is used to invoke an operation `m()` on a remote service. First, the pool of services conforming to `S` is retrieved. This pool is filtered, retaining those services that satisfy the pool membership condition (`constr`). Next, the remaining members are sorted according to the quality attribute provided by the `orderby` clause (`pref`). After performing these operations, `S2` turns out to be the service that best approximates the preferences of the developers, so `m()` is first invoked on that service. But the invocation fails due to a server error. Thus, the middleware transparently reinvokes `m()` on the next service in the sequence, `S1`, which successfully handles the call and returns a result.

### 6. A HIERARCHY OF TYPE QUALIFIERS

The extension of the Java type system with communication qualifiers for `multicast` (Section 4) and `reliable unicast` (Section 5) leads to the hierarchy of qualifiers shown in Figure 4. The root qualifier, `multi`, is an abstract qualifier. It acts as a stub pointing to an underlying collection of compatible services, without specifying the communication strategy. This abstract qualifier can be used to decorate variables that may refer to both groups and pools, thus allowing to change the invocation semantics of a variable at runtime. Each communication strategy is represented by a branch in the qualifier tree. The left branch contains the `pool` and `sequence` qualifier, which invoke an operation on any available endpoint. The right branch contains the `group` qualifier, which invokes an operation on all available endpoints.

![Figure 4: The hierarchy of type qualifiers](image)

**6.1 Changing Invocation Semantics**

The dotted arrows in Figure 4 show how method invocation semantics can be changed. We have already explained that the `where` operation is used to impose QoS constraints, and that a `pool` can be converted into a `sequence` with the `orderby` operation. Next to these behavioural changes, it is also possible to convert a `pool` into a `group` and vice versa.

Group variables, on one hand, are given the behaviour of a pool with the `any` operation. For example, the operation `(any g).m();` on a variable declared as `group G g` is invoked with reliable unicast semantics, even though `g` was declared as a group variable. Pool variables, on the other hand, are given multicast invocation semantics with the `all` operation. Thus, the operation `(all p).m()` on a variable declared as `pool P p` will multicast `m()` to all members of the underlying pool, even though `p` was declared as a pool variable.

**Evaluation.** The main advantage of integrating communication qualifiers with the Java type system is that the behaviour of a method invocation can be statically type-checked, which guarantees that a program is free of communication conflicts (O03). For example, because `sequence` is a subqualifier of `pool`, the compiler is able to guarantee that no unsorted sets of remote objects (pool) flow to places in the code where a deterministic selection policy (sequence) is required. Additionally, because `pool` and `group` are unrelated qualifiers in the hierarchy, the compiler can guarantee...
that pool references are never used to send multicast invocations, and vice versa. Indeed, this is only possible after the reference has been explicitly converted into the appropriate qualification (using any and all). This is done in order to avoid unintentional changes to the invocation semantics of qualified variables.

7. RELATED WORK

A large number of Java frameworks and language extensions have been proposed in order to extend Java with support for communication strategies other than the basic unreliable unicast mechanism of RMI. In this Section, we relate our proposal with (1) Java libraries and frameworks, (2) Java language extensions, and (3) approaches based on design patterns.

Libraries and Frameworks. Java libraries such as GMI [11] and ProActive [3] support multicast invocation and fault tolerance, but they force the programmer to explicitize the underlying server architecture. Using ProActive, for example, developers must write code for creating groups and adding group members, leading to a lower level of abstraction (OO1). Moreover, these libraries rely on a Meta-Object Protocol and inherit its limitations, such as the inability to invoke operations with a primitive return type or operations returning final classes.

Similar problems were found in OpusJava [10], a Java framework for distributed computing with support for remote object creation and asynchronous method invocation. The strength of OpusJava is that it allows to change the invocation semantics at the level of a single method invocation. It does so by offering methods such as call(methodName, arguments). By relying on reflection, however, OpusJava bypasses compile-time checks (OO3) on the method call. Indeed, by passing method names (methodName) as String objects, the compiler is unable to guarantee that the target object actually exports that method. Similarly, the compiler cannot check whether the argument list (arguments) conforms to the formal arguments of the method. A second problem with the OpusJava library is that both the return type and the arguments list are instances of Object, which disables sending arguments of a primitive type to and from remote objects. This reduces the interoperability of OpusJava applications with other RMI-based applications, which generally allow parameters of primitive types.

Filterfresh [2] is a Java package for building replicated, fault-tolerant servers based on an extension of the RMI registry. Similar to our approach, Filterfresh clients rely on specialized stubs that support reliable unicast. This behaviour, however, is hardcoded inside the stub, thus disabling complex interactions with the middleware (OO2). Our approach solves this problem by removing the hard-wired communication strategy from the stub. Instead, communication qualifiers are introduced to specify which invocation semantics must be used when calling a remote service.

Jini [8] is a Java framework with support for late binding and dynamic service discovery. Similar to our concept of constrained service pools, developers express constraints by programming templates. Such a template is sent to a discovery service and only services matching the constraints of this template are returned. This mechanism is weaker than our where clause because it only supports exact matches on predefined query instances. Another drawback of Jini is that developers must write a considerable amount of boilerplate code for service lookup and service selection (OO1) whereas our approach provides high-level abstractions for interacting with the middleware.

JavaSymphony [6] is a Java library that allows developers to control parallelism and object locality at a higher level than basic RMI systems. Using JavaSymphony, remote objects can be grouped into hierarchical domains, sites and clusters. One advantage of JavaSymphony is that it provides basic functionality for expressing constraints on a group of remote servers. This allows developers to declaratively specify where remote objects must be instantiated. Compared to our declarative where clause, this query mechanism is rather weak, however, because it supports only conjunctive queries on predefined constants such as the DNS name or the CPU load of a server. Another problem is that these queries are not type-safe (OO3) because the relational operators (<, <=, ... ) are represented as String instances.

Java Language Extensions. MChaRM [5] is a Java language extension based on reflective middleware. It refines communication channels and allows developers to fine-tune these channels similar to our language constructs. One problem with mChaRM is that it cannot provide static type checks on the communication semantics, due to its reliance on reflection (OO3). JR [9] is another Java extension for distributed programming, but it relies on hardwired references to physical service locations, thus limiting flexibility and reuse (OO2).

Design Patterns. Approaches based on design patterns typically reify a part of the distributed architecture, such as servers, ports, and communication channels. Gomaa et al. propose interconnection patterns [7] for programming client-server interactions. Using this approach, clients and servers communicate through connectors that hide the underlying communication infrastructure. Similar patterns for multicast and reliable unicast semantics are proposed by JavaGroups [4] and by the Object Group design pattern [12]. Both works are based on refined collections of replicated services. Compared to our hierarchy of communication qualifiers, which is defined in order to hide such collections, approaches based on design patterns provide a poor level of abstraction (OO1). Moreover, they force developers to instantiate a large number of pattern-related classes for dealing with technical middleware issues, thus obfuscating the business logic.

Our type system extension based on communication qualifiers resembles other type system extensions, such as [13], which introduces type qualifiers for creating transparent proxies for asynchronous method calls in Java, and Javari [14], which introduces type qualifiers for manipulating the accessibility of Java variables.

8. CONCLUSION AND FUTURE WORK

We have argued how communication qualifiers can be used to configure method invocation semantics in a declarative way. We have shown how the group qualifier is used to multicast method calls to a group of compatible services, and we have discussed how the pool and sequence qualifier are used for specifying a reliable unicast strategy. These communication strategies can be fine-tuned by means of declarative operations, such as where, orderby, all, and any.

Our type system extension enables programmers to imple-
ment different remote invocation semantics at a higher level of abstraction (OO1) using language constructs to fine-tune their communication strategy (OO2). Moreover, by integrating communication qualifiers with the type system, the compiler is able to detect communication conflicts by analyzing the flow of those qualifiers through a program (OO3).

We are currently working on a compiler prototype that transforms our language concepts to Java code. This guarantees interoperability with existing applications and it allows us to reuse the standard Java Virtual Machine for running applications enriched with communication qualifiers. Future work includes (1) a comparative study on how our concepts relate to and integrate with fragmented object systems, and (2) the specification and implementation of a more fine-grained model of communication qualifiers in order to support different communication strategies.

9. REFERENCES