MultiCaR: Remote invocation for large scale, context-aware applications

Gianpaolo Cugola
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
cugola@elet.polimi.it

Matteo Migliavacca
Department of Computing
Imperial College London, United Kingdom
migliava@doc.ic.ac.uk

***Abstract*—Remote Method Invocation (RMI) provides a powerful programming abstraction, well integrated with the object-oriented paradigm. Like conventional method calls, RMI interaction is point-to-point and uses an explicit address to determine the target of invocations. While natural and easy to use, these characteristics limit the applicability of RMI in large scale, dynamic scenarios. In this paper we present MultiCaR: a multicast extension to RMI, which provides a declarative addressing model that maximizes the decoupling among components, supporting a context-aware programming style that nicely fits dynamic scenarios. The set of guarantees provided by MultiCaR have been carefully defined to allow an efficient implementation of the model for large scale deployments. Finally, the MultiCaR prototype we developed exploits a content-based routing infrastructure to provide flexibility and scalability at the implementation level. We argue that these characteristics make MultiCaR a good candidate to develop large scale, object-oriented, dynamic applications, in the same way as content-based publish-subscribe has proved to support large scale, event-driven, dynamic applications.**

***Keywords*-Remote Method Invocation; declarative addressing; content-based routing; context-aware programming.**

I. Introduction

Advancement in telecommunication and information technology is establishing a new computation infrastructure being massively decentralized, strongly heterogeneous, and extremely dynamic. These characteristics demand unprecedented level of dynamism to distributed applications [1]. To answer this need, programmers make use of middleware abstractions that help them to build applications capable of operating in this new dynamic world. In particular, publish-subscribe has recently attracted programmers by supporting an interaction style among distributed components that is *multi-point*, *implicitly addressed*, and *asynchronous* [2], [3]. These characteristics provide the level of decoupling among components that allow them to evolve independently, adapting to a changing environment, and tolerating those failures that frequently happen in such dynamic scenarios.

On the other hand, as a natural extension to procedure call, remote method invocation (RMI) is still the most common paradigm to build object-oriented, distributed applications. Its characteristics are the opposite of those mentioned above: it is *point-to-point*, *explicitly addressed*, and *synchronous*.

While synchronous interaction is intrinsic in the concept of a remote method invocation, the other properties can in principle be removed, increasing the level of decoupling among components. In this paper we achieve this goal by designing MultiCaR: a multicast extension to RMI explicitly designed for large scale, dynamic environments. The MultiCaR model combines multicast invocation with a powerful, declarative addressing schema, which enables decoupling among distributed components.

The paper is structured as follows: in Section II we analyze the previous multicast RMI proposals, explaining why they are not suited to the scenarios we target. After a general description of such systems, we focus on their addressing schemas, recognizing their limitations. Section III presents our model of remote method invocation, while Section IV presents a scalable implementation of this model over REDS [4], a reconfigurable, distributed, content-based publish-subscribe infrastructure. Finally, Section V analyzes the performance of MultiCaR, while Section VI draws some conclusions and discusses future work.

II. Background and Motivation

The idea of building a new remote invocation originated from the need of simplifying the development of complex, context-aware applications for large scale, dynamic systems like those enabling pervasive computing. As mentioned in the previous section, our goal is to maximize the level of decoupling among the components that build such applications, leveraging a content-based routing infrastructure specifically tailored to dynamic scenarios, and to do so in the context of a remote invocation paradigm.

As an example of the applications we have in mind, consider an emergency response system designed to support the work of firefighters in the process of extinguishing a building on fire and evacuating the people inside. In this situation we could imagine that each firefighter is equipped with a PDA capable of communicating wirelessly with the control center, with nearby sensors, and with other firefighters' PDAs. Before entering a new area of the building the firefighter uses his PDA to access nearby sensors and *get the temperature from sensors within fifty meters in the north direction*, while the control center issues commands such as *alert all personnel not actually involved in evacuation procedures to leave the building immediately*.

Such kind of interaction is both very natural from a programmer's point of view and extremely powerful: it is multicast in nature and, most importantly, declaratively addressed. This allows for maximum decoupling between callers and callees. As an example, before the first invocation above takes place, the firefighter's PDA does not know the identity of the sensors around him, nor those sensors can anticipate that they will be invoked by some firefighter who happens to be nearby. It is the mutual "context" of the firefighter and sensors that determine the need of interaction.

Unfortunately, invocations of this kind are not supported by today's middleware: the multicast extensions to RMI proposed so far, in fact, were designed with totally different scenarios in mind and do not easily adapt to the large scale, loosely coupled, dynamic scenarios we envisioned above. This is particularly evident if we look at their binding and addressing model, which fail in providing the level of decoupling required by those scenarios. The remainder of this section motivates this claim by describing currently available multicast extensions to RMI.

A. Existing Multicast RMI Middleware

The remote invocation paradigm was originally born with the goal of easing the sharing of resources in the ARPANET by substituting all the application-specific request/reply protocols with a single one [5]. In 1988 Sun Microsystems submitted a specification [6] of their RPC implementation, which is still in widespread use. With the advent of the object-oriented paradigm, RPC was revisited into RMI by middleware such as CORBA and Java RMI. As a generalization of traditional method calls, RMI provides only unicast invocations (RPC had limited support for unreliable multicast invocations). This has been seen as a limitation and resulted in a number of proposals to extend RMI to multicast invocations, i.e., by letting the target of a specific invocation being a group of objects instead of a single one.

Interestingly, while RMI middleware were designed to support Internet-wide interactions, the research on multicast invocation styles focused on more controlled and smaller scale systems[1]. Multicast extensions to RMI were in fact proposed in two, different, domains: parallel programming; and fault-tolerant, replicated systems.

In the case of parallel programming systems, multicast messaging primitives, like MPI [7] and its Java derivatives [8]–[10] were used from the very beginning to disseminate a single request to a group of objects, letting each one compute and return a piece of the desired result. More recently, Group Method Invocation (GMI) [11] was proposed as a multicast extension to Java RMI, built on top of MPI, to bring the RMI programming style into the realm of parallel programming. Through the GMI services, clients can dispatch invocation calls to a group of targets and request either none, one, or all the results back, optionally aggregated on the way back to the caller through a *combine function*. groups and forbidding joins or leaves after the group is created [12]. This is not a severe limitation when running CPU-intensive, parallel applications on clusters, as the environment is mostly static and pretty well controlled, but it is instead totally unacceptable in the dynamic scenarios we are interested in, where new components come and leave in a very fluid situation.

Similar considerations hold for the second field where multicast invocations were proposed: that of fault-tolerant, replicated systems. In such domain, multicast RMI is used to replicate and keep synchronized several copies of an entity to increase dependability in the presence of faults. Middleware like Jgroup [13], JGroups [14], Filterfresh [15], and the system described in [16] aid the client programmer by providing invocation transparency on a group of replicated objects. In particular, each object in a group is assumed to be an exact replica of the others, invocations are broadcasted to all group members (too keep consistency), and a single result is returned to the client (all the members in a group are supposed to return the same result, so there is no reason to collect all of them). This ends up in a communication service characterized by a very specific semantics, which cannot be used as a general multicast invocation facility. Moreover, at the implementation level, a complex combinations of reliable, causal, and atomic broadcast between group members in a virtual synchrony setting has to be put in place to maintain the required replica consistency in the face of faults. This hampers scalability, which is limited to few replicas for each object.

B. Addressing in Existing Multicast RMI

In traditional RMI middleware, method invocation is mediated by a local representative for the remote entity called *stub*. The caller must obtain a stub for the target it want to invoke (usually through a lookup service), before using it to dispatch invocations. This behavior is also common in multicast RMI middleware, where each stub refers to a group of entities instead of a single one.

Moving from this observation, we introduce a first distinction among the various addressing schema used by multicast RMI systems by looking at the time when the targets of invocations are bound to stubs. We call *late-binding* a schema that allows the set of targets associated to each stub to change after the stub is created, *early-binding* a schema that does not offer this feature.

An early-binding approach usually performs better. It anticipates the resolution of targets, allowing them to be directly referred by the stub and used many times for different invocations, without additional costs for establishing the binding. This is the approach adopted by standard RMI and by the multicast extensions targeted at parallel systems, like GMI, which focus on performance. On the other hand,

[1]Probably due to the lack of an Internet-wide multicast infrastructure

an early-binding approach might raise consistency issues in dynamic environments when new targets become available and old targets leave or become unreachable due to network partitions. A late-binding approach is better suited to such scenarios. This is the choice adopted by multicast RMI systems designed to support fault-tolerant, replicated applications, which delegate precise membership computation to the replicated servers themselves.

Another dimension along which the addressing schemas used in multicast RMI systems can be categorized is the structure of the name space used to select the targets of the invocations. It can vary from simple, flat name spaces, as in the RMI registry, to structured name spaces, as in JNDI [17], up to complex naming systems in which names are composed of attribute-value pairs used by callers to lookup the targets for their invocations. This is the case of Jini [18], which allows remote entities to be retrieved according to the interfaces they implement and to the value of the attributes they specified at registration-time.

Most of the systems cited in the previous section, namely Jgroup, JGroups, GMI, and the system described in [16], use an explicit notion of "group" to which potential targets of invocations must explicitly join in combination with a flat name space to lookup groups. This approach has several limitations when used in large scale, dynamic scenarios. Indeed, flat names offers scarce flexibility because of their lack of expressiveness power, while the use of an explicit notion of "group" increases the coupling between application components by requiring an agreement between callers and callees on the grouping of entities.

To remove these limitations, MultiCaR provides a late-binding approach where the concept of group is never used explicitly. Callers specify the entities they want to reach through predicates over a set of attributes that define the relevant properties of each specific entity, while targets use a similar approach to select their potential invokers. The binding between callers and targets is determined by combining these two set of predicates at invocation-time. This schema, which is innovative in the panorama of multicast RMI extensions[2], provides the flexibility and decoupling among callers and callees that is required by the large scale, dynamic scenarios we target.

III. The MultiCaR Model

As mentioned above, in MultiCaR each entity (both invokers and targets) is characterized by a set of attributes expressing its relevant properties. As an example, meaningful properties for a firefighter are his location, his equipment, the current task he is involved, his role, and so on. More in general, these properties represent the contextual information associated to each entity. Accordingly, in the following we collectively refer to them as the *context* of the entity.

Each attribute building the context of an entity is composed of a triple $\langle name, type, value \rangle$. The *name* identifies the attribute, the *type* is a basic data type such as `string` or `integer`, and the *value* is the current value of the attribute. We say "current" because, just like in our example and in context-aware applications in general, the context of each entity may change and MultiCaR supports this by allowing the values of context attributes to change at run-time.

The context of each entity forms the basis for the powerful addressing model adopted by MultiCaR. When an entity needs to interact with others, either as a caller or as a callee, it specifies the context of the other endpoints through a *context-filter*, which is built as a set of *conditions*. Each condition is composed of a triple $\langle name, operator, value \rangle$. While *name* and *value* have the same meaning as above, the *operator* is chosen among a set of predefined operators including equality (applies to every type), prefix, suffix, and substring (apply to strings), and standard arithmetic comparison operators (apply to numeric types).

As an example of how context-filters can be used, consider the case of an entity that must act as a target for invocations. Upon creation its initial context is specified in the constructor; at export time[3] a context-filter is specified to implicitly select the callers that this entity is willing to serve; afterward, both the context and the context-filter may change to reflect changes in the entity state, including the set of callers the entity is ready to serve. Similarly, callers may specify a context-filter at stub-creation time to implicitly select the targets that will receive subsequent invocations. More precisely, each invocation involves:

- a couple $\langle C_{\text{invoker}}, CF_{\text{invoker}} \rangle$ with the first term expressing the context of the invoker and the second one the context-filter associated with the stub;
- a couple $\langle C_{\text{target}}, CF_{\text{target}} \rangle$ for each potential target. The first term represents the context of the target while the second one represents its context-filter (specified at export time).

A given target is actually chosen (and the invocation takes place) only if CF_{invoker} matches C_{target} and CF_{target} matches C_{invoker}. This check is performed at each invocation, resulting in a late-binding approach. In particular, the caller's context-filter CF_{invoker} is simply stored in the stub and the same happens to the context-filter of the targets CF_{target}, which is stored in the target's skeleton. At each invocation these filters, together with the current contexts of the invoker and targets, are used to determine the set of entities that are actually invoked. This gracefully accounts for possible changes that frequently happen in the application or in the external environment, in the large scale, dynamic scenarios we target.

To further support scalability, MultiCaR provides also a

[2]The Intentional Naming System [19] exploits a similar, albeit simpler, concept for message-based communication

[3]As in Java RMI, MultiCaR needs objects to be explicitly exported before they can be remotely invoked.

minimal set of guarantees. In particular, resembling what offered by conventional Java RMI, MultiCaR does not try to offer an exactly-once semantics, while simply providing the caller with information (in the form of an exception) about faults happened during invocation. When stubs are created, two alternative semantics can be selected: either an exception is returned if at least one target was not reachable or failed to complete the invocation, or the same exception is returned only if every target fails. By choosing among the two semantics, programmers may adapt the MultiCaR behavior to their needs, covering most of the cases that may happen. Moreover, this basic functionality can be used to build more complex, fault-tolerant behaviors, either at the application-level or as a higher-level middleware service.

IV. MULTICAR IMPLEMENTATION

In this section we describe how we implemented the MultiCaR model on top of REDS [4], [20], [21], our context-aware, reply-enabled, content-based publish-subscribe infrastructure. An important aspect of our implementation is that of being fully compatible with the original Java RMI: remote objects can be invoked both in unicast, via Java RMI, and in multicast, via MultiCaR, within the same application.

A. Background on Java RMI and REDS

In Java RMI each *remote object* is characterized by the *remote interfaces* it implements. Differently from standard objects, remote objects must be explicitly *exported* to allow remote invocation by clients. To do so, programmers can choose whether to extend the `UnicastRemoteObject` class or simply to invoke its static `exportObject` method. In both cases the object is inserted into the RMI runtime and is ready to accept clients' invocations.

The Java RMI architecture consists of three layers:

- the *stub/skeleton* layer is the interface between applications (client and server side) and the RMI system.
- the *remote reference* layer is responsible for the semantics of the invocation, e.g. unicast or multicast.
- the *transport* layer manages the low-level details, such as connection management and transmission of data.

REDS (REconfigurable Dispatching Service) [4] is a framework of Java classes developed at Politecnico di Milano to build large scale, highly reconfigurable, publish-subscribe systems. To support large scale scenarios, REDS provides the publish-subscribe service through a distributed network of *brokers*. Applications access this network using the `DispatchingService` interface, which exposes all the necessary methods to connect to a REDS broker, send and receive messages, or add new subscriptions. To support dynamic scenarios, REDS includes several ad-hoc mechanisms that automatically adapt the dispatching infrastructure when the underlying network changes (e.g., due to churn in a peer-to-peer scenario), efficiently restoring stale subscription information, and recovering lost messages.

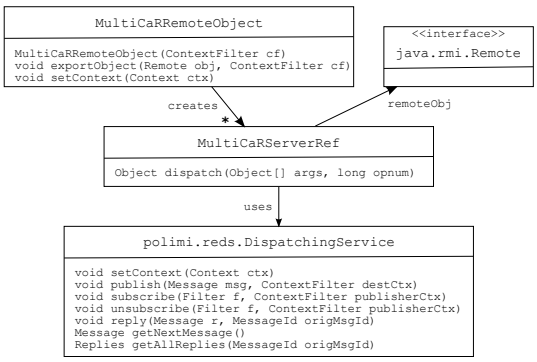

Figure 1. MultiCaR: main server-side classes and methods

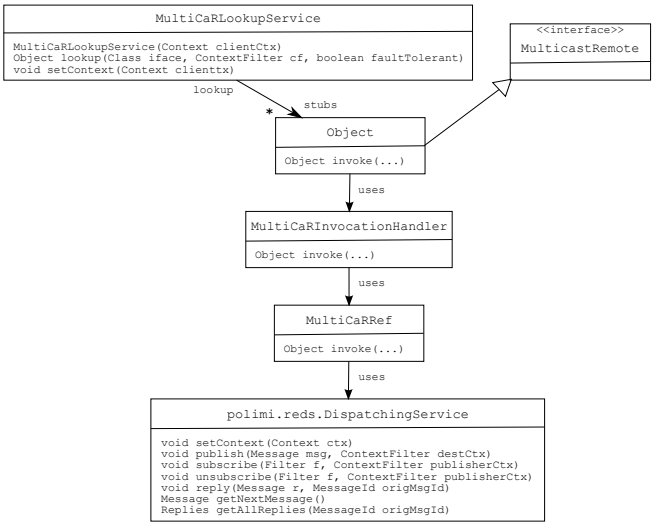

Figure 2. MultiCaR: main client-side classes and methods

REDS extends the traditional publish-subscribe model through an efficient, in-band mechanism to manage replies to received messages [20], thus naturally providing a bidirectional communication channel between publishers and subscribers. Recently, REDS has also been extended to support context-aware message routing [21]. Through the `DispatchingService`, clients can set their `Context`, while the methods to subscribe and publish have been augmented with a new parameter, the `ContextFilter`, used by an innovative, context-aware, routing schema to steer subscriptions only toward areas where matching context exists and to route messages only towards clients whose context matches the context-filter specified at publication.

B. The MultiCaR Prototype

The idea of implementing the MultiCaR model on top of REDS comes from the desire of exploiting the flexibility and scalability of the publish-subscribe paradigm to efficiently perform remote invocations in large scale, dynamic environments. Indeed, a context-aware, content-based, routing infrastructure as that realized by REDS, perfectly suit the complex needs that result from the MultiCaR addressing and

invocation models. The fact that REDS allows components to reply to the messages they receive, comes as a further benefit to ease the implementation of MultiCaR. Finally, the REDS ability of supporting changes in the topology of the routing infrastructure is crucial to cope with the dynamism of the scenarios we tailor.

The stub/skeleton layer. Moving from these premises, we may now analyze, starting from the stub/skeleton layer, how we actually used REDS to implement MultiCaR.

To create a multicast remote object, server-side developers may either extend the `MultiCaRRemoteObject` class or use its static `exportObject` method (see Figure 1). In the former case, the `MultiCaRRemoteObject` constructor takes the context-filter CF_{target} of the remote object as a parameter and calls the `exportObject` method. This method creates an appropriate `MultiCaRServerRef`, i.e., the skeleton for the remote object, and activates it.

The context C_{target} of the exported object can be set and changed at any time through the `MultiCaRRemoteObject.setContext` method. Similarly, the context-filter of the exported object can be changed at runtime (i.e., to change the set of clients it is willing to serve) re-exporting the object again.

On the client-side, an instance of the `MultiCaRLookupService` must be created passing the client's context C_{invoker} to the constructor (see Figure 2). Through this instance the client may create a stub to invoke a set of remote objects. This is obtained by calling the `lookup` method and passing it both the interface that the relevant remote objects must implement and the desired context-filter CF_{invoker}. The stub is dynamically created using the reflection services provided by Java, in the same way as the JDK 5.0 does for standard, unicast RMI. The stub delegates the actual invocation of remote methods to a `MultiCaRInvocationHandler`, which in turn uses a `MultiCaRRef` to invoke them.

Notice that the stub generated by the `MultiCaRLookupService.lookup` method implements an interface that differs slightly from the interface implemented by the relevant remote objects. It has the same methods of the original remote interface but differs in the return values: if the original interface's return value is of type `T`, the corresponding multicast interface method returns `T[]`. This is coherent with the multicast nature of this interface and allows the client to obtain all the return values provided by the invoked objects and collected by the MultiCaR runtime. Moreover, in a way similar to standard RMI, the newly created interface extends the tagging interface `MulticastRemote`.

As for the server-side, the client may change its context at any time by invoking the `setContext` method provided by the `MultiCaRLookupService` instance it uses. Conversely, to change the context-filter used to determine the set of relevant targets it must build a new stub. At a first sight this could seem to be a wasteful approach but this is not the case. In fact, besides the name, which has been chosen to resemble the standard RMI life-cycle, the `lookup` method does not need to contact any naming or directory service to build the required stub. All the information required to address the targets of future invocations (i.e., the `Context` and the `ContextFilter` instances used to determine the relevant targets) are simply stored into the returned stub and used dynamically by REDS, at each invocation, to determine the set of recipients, in the late-binding MultiCaR style.

The remote reference and transport layers. The two classes `MultiCaRServerRef` and `MultiCaRRef` mentioned above build the MultiCaR remote reference layer. The former is the skeleton for remote objects, in charge of receiving incoming calls and invoking the corresponding methods, while the latter operates on the client-side by dispatching calls to the relevant recipients. Instances of both classes use REDS as their addressing and transport layer.

In particular, the contexts and context-filters specified both by client and servers are passed to REDS through the `DispatchingService` instance used by the `MultiCaRRef` and `MultiCaRServerRef` objects, respectively, and they are used to route messages only toward the relevant targets.

More specifically, at the server-side, when the `MultiCaRServerRef` instance is created, it subscribes to messages published by clients whose context satisfies the remote object's context-filter and whose content includes one of the interfaces implemented by the remote object.

On the client-side, when a client calls a remote method through the stub, a new REDS message is created. It includes the interface the targets must implement, a numeric code that identifies the method to call, and the parameters to be passed to the target. This message is published by specifying the context-filter of the remote objects the client wants to reach. REDS uses the subscriptions issued by remote objects and the context information above to efficiently route this message only toward the relevant targets.

Finally, the converge-cast mechanism implemented by REDS to collect replies is used to transport return values back to the caller.

V. Evaluation

MultiCaR adopts a distributed, attribute-based name space with late-binding of targets; a combination that is unique in the panorama of RMI middleware and provides great expressiveness and flexibility. Consequently, a direct comparison of MultiCaR with other proposals is not possible. At the same time, we were interested in evaluating the effectiveness of MultiCaR, so we decided to focus on the performance and overhead that late-binding introduces w.r.t. the traditional early-binding approach of Java RMI, forgetting about expressiveness. Accordingly, in the following we assume that targets of interest could be identified by a predicate equivalent to a flat-name group selection.

Host name	Round-trip latency
planetlab6.csee.usf.edu	172.37 ms
planetlab7.csres.utexas.edu	153.24 ms
planetlab4.cs.uchicago.edu	126.99 ms
planetlab2.elet.polimi.it	684 μs
planetlab3.itwm.fraunhofer.de	21.22 ms
planetlab2.upm.ro	34.7 ms
plab3.ple.silweb.pl	36.44 ms
zoi.di.uoa.gr	45.94 ms
planetlab4.n.info.eng.osaka-cu.ac.jp	311.41 ms
plnodea.plaust.edu.cn	228.54 ms

Table I
PLANETLAB SITES.

	1 inv	10 inv	100 inv
Java RMI	2.61 s	4.27 s	21 s
MultiCaR	0.403 s	4.03 s	40 s

Table II
LATENCY FOR A SEQUENCE OF INVOCATIONS ON 5 TARGETS, 1 SITE.

In our tests we measured the latency of a multicast invocation to a group of targets. Targets were hosted on *sites*: workstations in the PlanetLab network selected to cover a wide geographical area. A *measuring site*, located at Politecnico di Milano, started the invocations and recorded the latency to collect replies. Table I reports the 10 selected *sites* with their latencies from the measuring site. Each target was exported and registered in the local RMI Registry and in the local REDS broker.

Invoking a set of distributed targets using Java RMI requires two operations: looking up the targets on the RMI registries and binding the corresponding stubs before invocation. Performing these operations and the following invocations sequentially would be strongly inefficient, thus we used multiple threads provided by the invoker. We used one thread per site for looking up the targets and one thread per target to perform the initial binding and the following invocations. In the case of MultiCaR, invocations were performed by creating a context-filter choosing the required targets and then issuing the call, which sends the invocation message to the REDS broker local to the invoker and from there to the other sites (each having its own broker).

Our servers defined a simple public method accepting two parameters each composed of three fields: a `String`, a `long`, and a `double`. To measure more accurately the communication costs the tested methods runs quickly, performing a simple comparison and returning one of the parameters to the invoker. Each test was repeated 30 times to calculate and plot the median across the runs. Notice that we expect that increasing the size of the parameters will have a higher impact on Java RMI than on MultiCaR, which optimizes multicast dissemination by forwarding a single invocation per site instead of one invocation per target.

Single site overhead. Table II shows the latency to invoke, respectively 1, 10 and 100 times, five targets deployed on the topmost site of Table I. The latency to perform a single invocation is substantially higher in Java RMI than in MultiCaR (more than 6 times). Indeed, by the time Java RMI has completed the first invocation MultiCaR was able to perform more than 10 invocations. Only if the client performs more invocations *on the same set of targets* the latency of Java RMI is better (i.e., as it approaches 100 invocations).

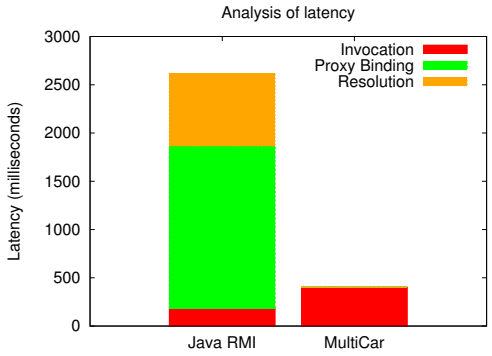

Figure 3. Breakdown of invocation latency for 5 targets, 1 site.

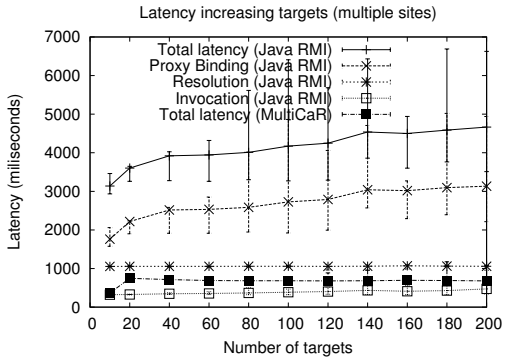

Figure 4. Breakdown of invocation latency growing the number of targets, 10 sites. Figure shows minimum, median and 90^{th} perc. of latencies.

Figure 3 reports the individual contributions to the latency for the first invocation. This includes the time for binding targets and the time for invoking them. The latter is smaller for Java RMI than for MultiCaR, but invocation in MultiCaR includes binding, while the two phases are disjoint in Java RMI. In a dynamic scenario, in which late-binding is required and consequently Java RMI has to re-bind objects at each invocation, the total cost is the important one, and this is where MultiCaR shines.

Multisite performance. Figure 4 reports the results of a distributed test involving all selected sites. The chart shows how latencies vary when increasing the total number of object invoked from 10 to 200, split evenly across the ten sites. We see that while the latency for looking up the targets remains constant (the number of sites is constant) the latency to bind the stubs increases with the number of targets. Finally, the overhead of MultiCaR with respect to Java RMI does not grows significantly with the number of targets, a sign of good scalability.

In conclusions, we can say that MultiCaR outperforms Java RMI in all those scenarios where a late-binding ap-

proach is required, including the "mixed" scenarios where it is enough to perform one binding every 10 invocations. For static scenarios, the early-binding approach of Java RMI performs better.

VI. Conclusions

In this paper we presented MultiCaR, a new middleware for multicast remote method invocations, designed to support large scale, dynamic, context-aware applications.

By adopting an innovative, late-binding, attribute-based addressing schema, MultiCaR provides an unprecedented level of expressiveness, which maximizes decoupling among components, supporting dynamic applications capable of changing their architecture at run-time.

At the implementation level MultiCaR supports scalability and dynamism thanks to the characteristics of REDS, the distributed publish-subscribe substrate on top of which it is built. Indeed, REDS offers efficient dispatching of data in large scale scenarios, providing mechanisms to tolerate and mask reconfigurations of the underlying network (e.g., to operate in mobile, wireless or in dynamic peer-to-peer scenarios).

Our future plans include using MultiCaR as a substrate to implement a higher level coordination infrastructure based on the concept of Distributed Abstract Data Type [22].

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects FIRB INSYEME and PRIN D-ASAP.

References

[1] L. Baresi, E. D. Nitto, and C. Ghezzi, "Toward open-world software: Issue and challenges." *IEEE Computer*, vol. 39, no. 10, pp. 36–43, 2006.

[2] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*. Springer, 2006.

[3] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[4] G. Cugola and G. Picco, "REDS: A Reconfigurable Dispatching System," in *Proc. of the 6th Int. Workshop on Soft. Eng. and Middleware*. Portland, Oregon, USA: ACM Press, nov 2006, pp. 9—16.

[5] J. White, "High-level framework for network-based resource sharing," RFC 707, Dec. 1975.

[6] S. Microsystems, "RPC: Remote Procedure Call Protocol specification: Version 2," RFC 1057, Jun. 1988.

[7] M. P. I. Forum, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, 1994.

[8] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, "Mpi-java: An object-oriented java interface to mpi," in *IPPS/SPDP Workshops*, ser. LNCS, vol. 1586. Springer, 1999.

[9] G. Judd, M. J. Clement, and Q. Snell, "Dogma: distributed object group metacomputing architecture," *Concurrency - Pract. and Exp.*, vol. 10, no. 11-13, pp. 977–983, 1998.

[10] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, "Mpj: Mpi-like message passing for java," *Concurrency - Pract. and Exp.*, vol. 12, no. 11, pp. 1019–1038, 2000.

[11] J. Maassen, T. Kielmann, and H. Bal, "GMI: Flexible and efficient group method invocation for parallel programming," in *Proc. of the 6th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, 2002.

[12] A. Nelisse, T. Kielmann, H. E. Bal, and J. Maassen, "Object-based collective communication in java," in *Proc. of the 2001 joint ACM-ISCOPE Conf. on Java Grande*. NY, USA: ACM Press, 2001, pp. 11–20.

[13] A. Montresor, "The jgroup reliable distributed object model," in *Proc. of the 2nd IFIP Int. Working Conf. on Dist. Appl. and Int. Sys.* Helsinki, Finland: Kluwer, Jun. 1999, pp. 389–402.

[14] B. Ban, "Design and implementation of a reliable group communication toolkit for java," Cornell University, Tech. Rep., 1998.

[15] A. Baratloo, P. E. Chung, Y. Huang, S. Rangarajan, and S. Yajnik, "Filterfresh: Hot replication of java rmi server objects," in *COOTS*. USENIX, 1998, pp. 65–78.

[16] W. C. Massimo, "Enhancing java to support object groups," in *ROOTS'02*, 2002.

[17] JNDI Website, http://java.sun.com/products/jndi/.

[18] Jini Website, http://www.jini.org/.

[19] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," in *SOSP*, 1999, pp. 186–201.

[20] G. Cugola, M. Migliavacca, and A. Monguzzi, "On adding replies to publish-subscribe," in *DEBS*, 2007, pp. 128–138.

[21] G. Cugola, A. Margara, and M. Migliavacca, "Context-aware publish-subscribe: Model, implementation, and evaluation," in *Proc. of the IEEE Symp. on Comp. and Comm.*, Sousse, Tunisia, July 2009.

[22] G. P. Picco, M. Migliavacca, A. L. Murphy, and G.-C. Roman, "Distributed abstract data types," in *OTM Conferences (2)*, ser. LNCS, vol. 4276. Springer, 2006, pp. 1594–1612.