Enhancing Java RMI with Asynchrony through Reflection

Orhan Akın, Nadia Erdoğan

Istanbul Technical University, Computer Sciences, Informatics Institute,
Maslak-34469, Istanbul, Turkey
{orhan}@.engineer.com, {nerdogan}@itu.edu.tr

Abstract. Java RMI's synchronous invocation model may cause scalability challenges when long duration invocations are targeted. One way of overcoming this difficulty is adopting an *asynchronous* mode of operation. An asynchronous invocation allows the client to continue with its computation after dispatching a call, thus eliminating the need to wait idle while its request is being process by a remote server. This paper describes an execution framework which extends Java RMI functionality with asynchrony. It is implemented on top of RMI calls, using the thread pooling capability and the reflection mechanism of Java. It differs from previous work as it does not require any external tool, preprocessor, or compiler and it may be integrated with previously developed software as no modification of target remote objects is necessary..

Keywords: Asynchronous Communication, Asynchronous RMI, RMI, Reflection, parallel programming, distributed programming.

1 Introduction

Communication is a fundamental issue in distributed computing. Middleware systems offer a high level of abstraction that simplifies communication between distributed object components. Java's Remote Method Invocation (RMI) [1] mechanism is such an abstraction that supports an object-based communication framework. It extends the semantics of local method calls to remote objects, by allowing client components to invoke methods of objects that are possibly located on remote servers. In RMI, method invocation is synchronous, that is, the operation is blocking for the client if it needs the result of the operation or an acknowledgement. This approach generally works fine in LANs where communication between two machines is generally fast and reliable [2]. However, in a wide-area system, as communication latency grows with orders of magnitude, synchronous invocation may become a handicap. In addition, RMI's synchronous invocation model may cause scalability challenges when long duration invocations are targeted. One way of overcoming these restrictions is adopting an asynchronous mode of operation. This type of invocation provides the client with the option of doing some useful work

while its call request is being processed, instead of being blocked until the results arrive. Asynchronous invocations have the following advantages:

- to overlap local computation with remote computation and communication in order to tolerate high communication latencies in wide-area distributed systems,
- to anticipate the scheduling and the execution of activities that do not completely depend on the result of an invocation,
- to easily support interactions for long-running transactions
- to enforce loose coupling between clients and remote servers.

We propose an execution framework which extends RMI functionality with asynchrony. Clients are equipped with an interface through which they can make asynchronous invocations on remote objects and continue execution without the need to wait for the result of the call. They can later on stop to query the result of the call if it is required for the computation to proceed, or they may completely ignore it, as some invocations may not even produce a result.

Our framework focuses on the four most commonly used techniques, namely fire and forget, sync with server, polling, and result callback for providing client-side asynchrony. The design of the framework is based on a set of asynchrony patterns for these techniques described in detail in [3]; actually implementing the asynchrony patterns on top of synchronous RMI calls.

This paper describes the design and implementation issues of the proposed framework. Java's threading capability has been used to provide a mechanism for asynchronous invocation. One contribution of this work is its use of run-time reflection to do the remote invocation, thus eliminating the need for any byte code adjustments or a new RMI preprocessor/compiler. Another contribution is the rich set of asynchronous call alternatives it provides the client with, which are accessible through an interface very similar to that of traditional RMI calls. As the framework is implemented on the client side and requires no modification on server code, it is easily integrated with existing server software.

2 Java RMI

Java RMI provides a remote communications mechanism between Java clients and remote objects [1]. Java clients obtain references to these remote objects through a third party registry service. These references allow transparent access to the remote object's methods by mirroring the remote interface. RMI hides all execution details of communication, parameter passing and object serialization details by delegating them to 'stub' objects at both sides, on the client and the server. Method invocation is synchronous. This mode of communication may be satisfactory in applications where execution time is not critical. However, many distributed applications may benefit asynchronous invocations.

3 Asynchronous Invocation Patterns

Asynchronous invocation allows the client to continue with its computation after dispatching a call, thus eliminating the need to wait idle while its request is being process by a remote server. Several alternatives exist on how the results are passed to the client. The client may be interrupted when the results are ready, it may receive the results when it needs them, or it might not even be interested in the result. Multiple method invocations may as well be interleaved, without retrieving the response in between. There are four most commonly used techniques for providing client-side asynchrony [3].

Fire and Forget: well suited for applications where a remote object needs to be notified and a result or a return value is not required.

Sync with Server: similar to fire and forget; however, it ensures that request has been successfully transferred to the server application.

Polling: suitable for applications where a remote object needs to be invoked asynchronously, and yet, a result is required. However, the client may continue with its execution and retrieve the results later.

Result Callback: similar to Poll Object, a remote operation needs to be invoked asynchronously and a result is required. The client requests to be actively notified of the returning result

4. Design Objectives for Asynchronous RMI Execution Framework

- **Independence of any external tools, preprocessors, or compilers:**

Our main objective has been to present an execution framework that is completely compatible with standard Java, compilers, and run-time systems and does not require any preprocessing or a modified stub compiler. We have used RMI as the underlying communication mechanism and implemented asynchrony patterns on top RMI calls (as depicted in Figure 1.). Therefore, as long as a client is able to access a remote object using standard RMI, both invocation models, asynchronous /synchronous invocations, are possible. The main benefit for the developer is that he may choose the appropriate invocation model according to the needs of the application.

- **No modification of existing server software**

Another design objective is that the framework should require no modifications on the server side so that previously developed remote objects can still be accessed, now asynchronously as well. There is no necessity for a remote object to implement a particular interface or to be derived from a certain class as the framework is located on the client side.

- **Performance related concerns**

Performance issues have also been the focus of the implementation. As threading and reflection produce extra runtime overhead, special care has been taken to keep their use at minimum, so that they do not introduce a performance penalty on method execution

5 Execution Framework Implementation

The asynchronous RMI execution model has been implemented by an execution framework (Figure 1.) which is developed in Java. It consists of a Java package itu.rmi.* containing the classes that provide the basic services for asynchronous invocations. The classes that are visible to the client are depicted in Figure 2 . The execution flow of an asynchronous invocation (arrows 1 through 7 in Fig. 1.) and its implementation details are described below, assuming that a server has already registered a remote object with an RMI registry, making it available to remote clients.

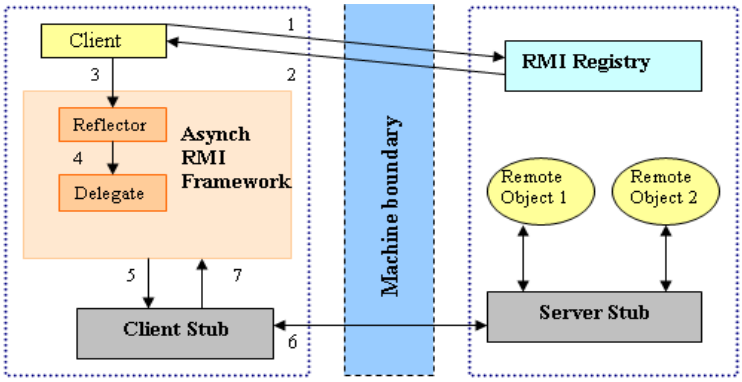

Fig. 1. Asynchronous RMI Framework Execution Flow

Initialization Phase:

1. This step involves the determination of the remote host. The client queries the lookup service *rmi registry* to retrieve a remote reference to the target remote object.
2. The lookup call returns a remote reference and results in the placement of a client stub which provides the interface to interact with the remote object.

Asynchronous Invocation Request:

3. An asynchronous invocation requires an instance of an invocation object to be created for the specific type of asynchronous call *(FireandForget, SyncWithServer, Polling, Callback)* the client wishes to make. The asynchronous call is dispatched as the client invokes the public void call(Object remoteObject, String methodName, Object params[]) overloaded call method of the invocation object with the actual parameters that specify the client stub reference, the name and the list of parameters of the remote method to be called. The client resumes execution as soon as the call method returns.

Asynchronous Invocation Processing:

4. The call method receives an invocation request and calls the appropriate method on the remote object transparently, applying the semantics of the specific type of asynchronous call. For this purpose, it uses reflection to assemble a method call during runtime. Once the method name and the argument types are resolved, they are matched with those of the incoming

request to detect errors such as invoking a non-existing method, passing an incorrect number of arguments, or passing incorrect argument types to a method, raising exceptions that are caught by client. If no error exists, the call method returns after starting a service thread which handles the remainder of the invocation, allowing the client to resume execution while the asynchronous call proceeds in the new service thread. The service thread, called a *delegate* in the our framework, is activated from a thread pool with parameters that include invocation details such as the remote object, method name, and parameters.

5. The delegate simply executes an invoke method call in its run method, (_remoteMethod.invoke (_remoteObject,_parameters);) which performs a standard RMI for the requested method over the client stub.
6. This phase involves standard RMI activity following its parameter passing semantics. The server stub passes the call request with its parameters to the server stub, which in turn, executes the method call on the remote object and returns the results back to the client stub over the network.
7. The client stub returns the result of the remote call to the *delegate*. The delegate responds differently on retrieving the result, according to the specific type of asynchronous call.

6 Types of Asynchronous Calls

Fig. 2 displays the public classes visible to user programs.

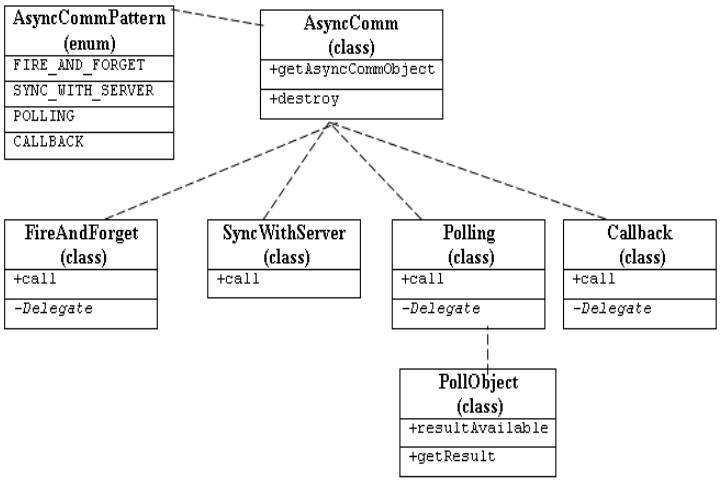

Fig. 2. Execution Framework Public Classes

The execution framework supports the four most commonly used asynchronous calls. In the following sections, we describe their execution patterns and give code samples to show how to utilize the framework for such calls, assuming the existence

of the remote server objects logger, docConverter, searchEngine, and Emergency on a host with IP '192.168.1.3'.

Fire and Forget: Fire and forget is well suited for applications where a remote object needs to be notified and a result or a return value is not required. Reliability is not a concern for both sides. When the client issues such an invocation request, the call returns back to the client immediately. The client does not get any acknowledgment from the remote object receiving the invocation.

The client invokes asynchronously the log method of the remote object logger with the string parameter "my log message". The call method returns right after dispatching the delegate with the call request. Thus, the client thread and the delegate thread run concurrently, the client returning back to its execution while the delegate blocked waiting for the call to return.

```
fireAndForget.call(logger, "log", new Object[] {"my log
message"});
```

Sync with Server: Sync with Server is similar to fire and forget; however, it ensures that the invocation has been performed reliably. Again, a remote object needs to be notified and a result of the remote computation is not required. The difference is that, the call does not immediately return to the client but waits for an acknowledgement from the remote object to ensure that the request has been successfully transferred to the server application. Only then, control passes to the client. Meanwhile, the server application independently executes the invocation.

The client invokes asynchronously the convert method of the remote object docConverter with the input parameter msDoc. The call method blocks until the server application docConverter returns an acknowledgement that it has successfully received the request. From that point on, both the client and the server in parallel, the client returning back to its execution while the docConverter proceeds with the document conversion process.

```
boolean b = ((Boolean)syncWithServer.call(docConverter,
"convert",new Object[] { msDoc })).booleanValue();
```

Polling: Polling is suitable for applications where a remote object needs to be invoked asynchronously, and yet, a result is required. However, as the results may not be needed immediately for the client to proceed with its computation, the client may continue with its execution and retrieve the results later. In such a case, a poll object receives the result of remote invocations on behalf of the client. The client, at an appropriate point in its execution path, uses this object to query the result and obtain it. It may poll the object and continue with its computation if the result has not yet arrived, or it may block on the object until the result becomes available.

The client issues a polling call to the search method with the keyword "java" as the parameter. This time, the method call returns with a poll object as soon as it dispatches the delegate with the call request, allowing the client to continue with its execution while remote object processes the query and produces a result. When a result is returned, the delegate thread retrieves it in the poll object, sets a flag in the object to indicate that it is available and notifies any thread blocked on the object. The resultAvailable() method of the poll object returns a boolean value and may be checked by the client to see if a result has arrived. The client may also call the

getResult()method of the poll object to get the result. However, this is a blocking call and does not return until the result actually becomes available.

```
PollObject pollObject= (PollObject) polling.call
(searchEngine, "search", new Object[] {"java"});

boolean b = pollObject.resultAvailable();

List<String> result = (List<String>)
pollObject.getResult();
```

Callback: Similar to Polling, a remote operation needs to be invoked asynchronously and a result is required. However, in this case, the client needs to react to the result immediately it becomes available, not at a future time of its choice. Therefore, the client requests to be actively notified of the returning result. To this end, the client passes a callback object together with the invocation request to the execution framework. The call returns after sending the invocation to the server object and the client resumes execution. Once the result becomes available, a predefined operation on the callback object is called, passing it the result of the invocation.

Our implementation makes use of the Observer Pattern [5], therefore, callbackObject, an instance of a class which implements Observer interface is created before the client makes a call to the sendEmergency method, supplying a criterion "pressure is below 10" and a callback object callbackObject. Once the result of the remote call becomes, the update method of the observer object is invoked automatically with result parameters by the delegate using built-in method notifyObservers.

```
CallbackObject callbackObject = new CallbackObject();

callback.call(emergency, "sendEmergency", new Object[]

    {"pressure is below 10"}, callbackObject);
```

7 Performance Optimization

Performance of the framework has been a central concern during implementation. We tried to determine the greatest sources of runtime overhead and observed them to be related to threading facility and reflection mechanism of Java. Below, we describe the optimizations we have done.

Thread Pool: The framework transfers each new asynchronous invocation request to a new thread. Considering the overhead of spawning a new thread on each call, we optimized the interaction with the thread package and switched to using a ThreadPool which creates new threads as needed, but reuses previously created threads when they are available. Fig. 3. displays the performance gain in invocation response time, which becomes evident especially with high number of simultaneous requests.

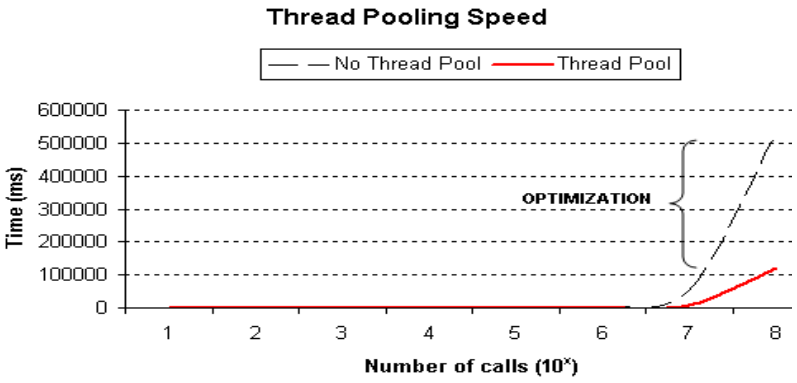

Fig. 3. Thread Pooling Speedup

Reflection: The second optimization we performed concerns reflection. When the call method of an invocation object receives an invocation request, it uses reflection to resolve the method name, the number of its arguments and their types. To minimize the overhead introduced by reflection, we cash the recently resolved information in private fields of the invocation object so thatsuccessive requests for a particular invocation instantly use the local data, instead of reflection lookups. Figure 4. displays speed up gained through reflection optimization.

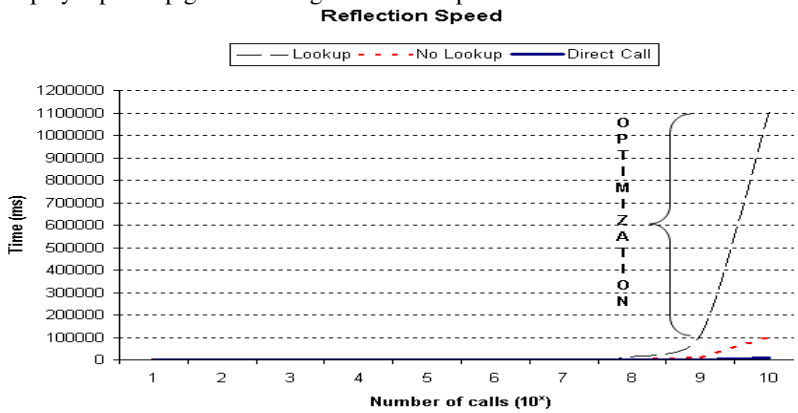

Fig. 4. Reflection speedup

Performance Results:

We have conducted experiments in order to measure the performance of the framework and compare it with Java RMI. The test computer is Intel Core 2 Duo CPU T8300 2.4 GHz, 2GB of RAM, Windows XP Professional OS, Java build 1.6.0_11-b0. and the remote computer is Intel Pentium 4 Mobile CPU 1.70GHz, 512 MB of RAM. Java build 1.6.0_11-b03, Windows XP Professional OS.

Table 5. displays the execution cost for an asynchronous invocation of a remote method with the signature 'int add(int lhs, int rhs)' that simply adds the two input parameters and returns the result. For each invocation type, five test runs were

executed and their average is reported. *Call Time* is the elapsed time for the client to make the call and resume execution. *Result Retrieval Time* is the time it takes for the result to be available. For polling type of invocation, this is the time duration until the blocking call getResult returns. For Callback, it is the duration until the update method of the observer object gets invoked.

Table 1. Asynchronous invocation execuiton costs

	Fire and Forget	Sync With Server	Polling	Callback
Call Time (ns)	3901948	11767026	3126207	2879528
Result Retrieval Time (ns)	NaN	11767026	4229923	4486045

We have implemented the remote objects introduced in Section 6 to demonstrate the different types of asynchronous calls and executed both standard RMI and asynchronous RMI calls to compare their execution time values. We assume that it takes 400 ms for the remote search engine to find a given keyword and a criterion is met every 2200 ms at the remote emergency object. Figure 6 illustrates the results, where the idle waiting state of the client issuing a standard RMI is clearly seen.

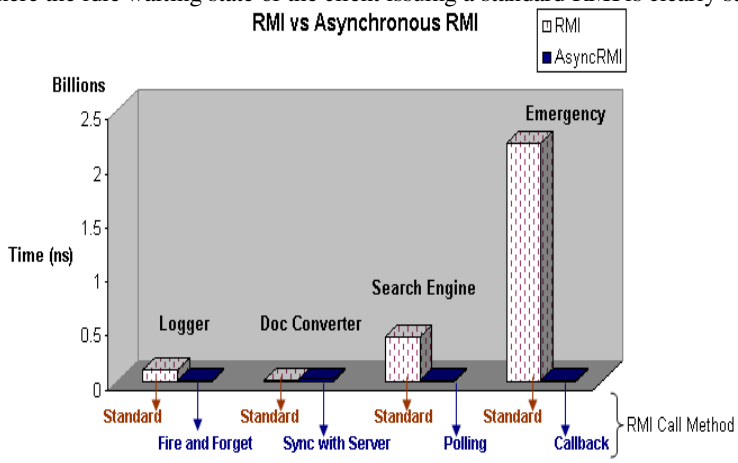

Fig. 6. Standard RMI versus Asynchronous RMI

8 Related Work

Asynchronous RMI has been the focus of several projects in literature, which usually focus only on selected aspects of the RMI, like communication performance, asynchronous invocation, or interoperability. [4] presents a system which uses the concept of a future object for asynchrony. A special compiler (armic) is used to generate stubs that handle asynchronous communication. The stubs provide asynchronous communication by wrapping each remote invocation inside a thread. Server side of the connection must also be threaded; therefore the system can not be

used with remote objects already created. Furthermore, their approach is different from ours in that they do their callbacks directly from the server. In comparison, our callbacks are local. The Reflective Remote Method Invocation (RRMI) [5] is close to our approach as it makes use of reflection and provides a mechanism for asynchronous invocation. RRMI makes use of a dynamic class loader which is an extension to Java class loader (NetClassLoader) to allow client/server applications to be built. The NinjaRMI project [6] is a completely rebuilt version of RMI with extended features, including asynchronous communication. However, it is not wire compliant with standard Java RMI. It is intended to provide language extensions. The Ajents project [7] is a collection of class libraries that implement and combine several features that are essential for distributed computing using Java. However, its asynchronous communication aspect does not provide wide functionally; only the Polling pattern is implemented partially using 'future' objects.

9 Conclusion

This paper presents an execution framework which extends Java RMI to support asynchronous communication, mainly focusing on the techniques of fire and forget, sync with server, polling, and result callback. We have used RMI as the underlying communication mechanism and implemented asynchrony patterns on top of RMI calls. The threading facility and reflection mechanism of Java has made it possible for the framework to be independent of any external tools, preprocessors, or compilers. Also, as the framework requires no modifications on the server side, therefore previously developed remote objects can still be accessed asynchronously. The results of performance measurements show that, with optimizations on threading and reflection activity, the enhanced asynchronous RMI communication we have described in this paper produces a dramatic performance increase by removing unnecessary delays caused by blocking on synchronous RMI invocations.

References

1. Sun Microsystems, Inc. Java (TM) Remote Method Invocation Specification, (2004)
2. Andrew S. Tanenbaum and Maarten Van Steen, Distributed Systems: Principles and Paradigms, Pearson-Prentice Hall, ISBN:0-13-239227-5
3. M. Voelter, M. Kircher, U. Zdun, and M. Englbrecht, "Patterns for Asynchronous Invocation in Distributed Object Frameworks," in The 8th European Conference on Pattern Languages of Programs (EuroPlop 2003), Irsee, Germany, 2003, pp. 269-284, (2003)
4. R. Raje, J. I. William, and M. Boyles. An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. In ACM 1997 Workshop on Java for Science and Eng. Comp., (1997)
5. George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. "Reflective Remote Method Invocation", Concurrency: Practice and Experience, 10(11- 13):911-926, (1998)
6. Matt Welsh. Ninja RMI : A Free Java RMI. Available from http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html, (1999)
7. M. Izatt, P. Chan, T. Brecht, Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications , Concurrency: Pract. & Exper; 12:667–685, (2000)