

Review of RMI Architectural document

Stijn van Drongelen
María Hernández
Johannes Leupolz
Alejandro Serrano

1 General considerations

Java Remote Method Invocation, or short Java RMI or even RMI, has a very specific aim: it allows applications to access and execute methods in objects on a remote server. It is meant to be used in a very transparent way: Users do not need to write code for networking or (de-)serialisation - they just write their implementation and interfaces which should be accessible from other machines and RMI takes care about the remoteness[2]. Therefore RMI exploits features of the Java Virtual Machine. As the paper under our scope[1] explains, this is done inside RMI by mean of sockets. Local stubs are created by RMI and a coder can invoke its methods as if the objects were local. Those stubs are the ones which takes care of the communication and serialization.

From our point of view, after a reading of [1], the RMI architecture is made up by the parts shown in Figure 1. Those components are the ones that are going to become important when doing a deep analysis of the system.

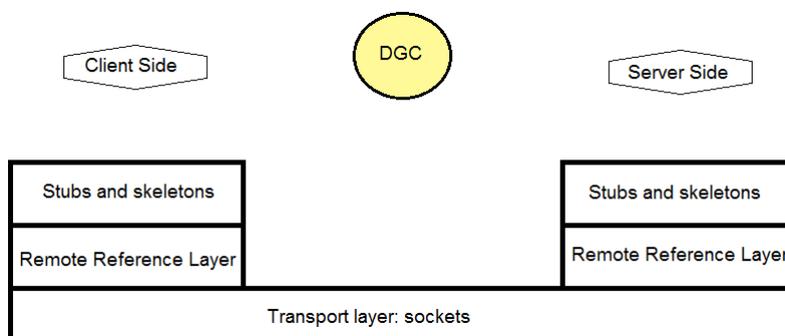


Figure 1: Components in a RMI system

The use of RMI is quite simple and clients and server must follow a quite strict set of steps in order to establish a RMI connection. Because of that, if they want to introduce some changes in the way RMI is used, adding or changing some functionality, this must be done by writing Java code out of the RMI mechanism. Thus, RMI can cope with almost every situation, as long as the coder writes a factory which creates a class with the desired features. For instance, as stated in the architectural document, encryption, authentication and transactions are not covered *by default* by RMI, but they can be added by using a different socket factory.

In this paper, we will consider some changes in the requirements of RMI and analyse how the system must change in order to satisfy those new requirements. First, we will deal with a change which leads to small changes in the architecture, and later with two other changes that will require the architecture to change more. For each, we will follow some of the items in the *Impact Analysis Checklist*[4]. At the end, we will analyse whether the architectural document[1] that we are reviewing is enough for knowing the changes that must undergo in the architecture, or we have needed other sources of information.

2 Changes in the requirements with a small effect on the architecture

The use case to consider here is illustrated in figure 2:

*The RMI server and the RMI client are in different local networks.
Only Webservers in these network have access to the Internet*

That is, we would like to be able to use a new way of transporting the information because we want to add a new layer of security on the protocol -for example, checking LDAP credentials- or want to bypass some new proxy.

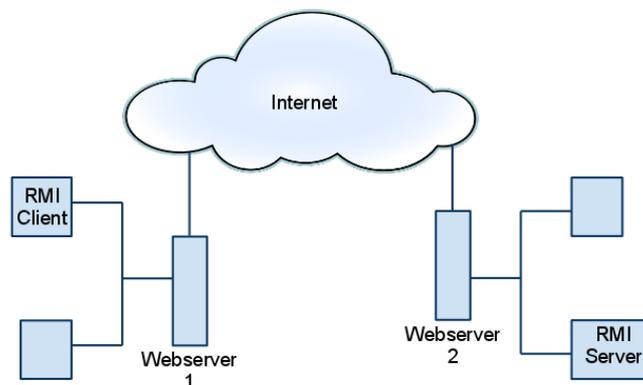


Figure 2: Small change: only webservers in the network have access to the Internet

Need for the change This change may arise from the necessity of an organization of introducing some changes in its security policies, such as a strong encryption.

Consequences of not making the change First, missing the advantages that using the new protocol can give to the organization: more security, better authentication... this is the reason why the change can be introduced, but there can also be other consequences. In the worst case, inability to contact the server we want to talk to: if RMI doesn't allow this kind of connection, the client could not contact with the object in the server side. Besides, we would need to use other protocol because of extra measures in our network; if we do not do it, maybe we need more network management work force (adding proxies, putting things in another network segment...).

Risk of making the change A bad implementation may make the RMI application crash, even though the system should not break all: the problem can be treated as a

communication problem, for which RMI has protective measures now. We should also take care of security risks, as always that changes are introduced in a protocol. This can be avoided by using an already-available implementation (which Java will have for sure) instead of implementing a new protocol by ourselves. If it exists, it will only be necessary to write the part to adapt it to the application.

Java now allows different types of connections. This lead us to think that the info or RMI objects and systems is hold in a common way, so we do not need to take care of information travelling the net. And, in principle, no worries about performance loss.

Conflicting requirements Is there any conflict when this requirement is introduced with the previous ones? RMI follows an order in the ways it tries to connect to the server¹, with this change we want to modify that order. But, the main aspect of conflict is with the sockets: connections will always consist of Sockets, and the user can only change the socket factory; but we may want to use a better-suited IPC mechanism, such as D-Bus. In this case, changes in the architecture must be introduced. However, maybe this can be done just by using a Stream interface instead of the Socket one.

Moreover, you can only change the socket factory once in your application [5] -the second time an IOException is thrown-, so since the beginning the server must decide what protocol is going to be used to connect to it.

Things that are good in the architecture regarding this point We just want to extend RMI by been used in a more general protocol, so we do not need to know the details of RMI protocol, just wrap it in ours. This is a very good point in the way RMI is done, since it has not to be changed, there is less risk of missing something in it.

About the Stream interface mentioned before, RMI could work with both approaches: sockets and streams. It would only be necessary to establish a class hierarchy for that. This may leads to the necessity of parsing RMI messages in order to distinguish which protocol they have been sent by.

Affected parts According to the paper that we are reviewing, the changes for this new requirement must be done in the Transport Layer (see 1.2.2). This layer is based on TCP/IP connections and provides mechanisms to connect to the host machine and basic strategies to go through the firewall. The changes related to the way of connecting must be introduced here. In the document, three ways of breaking through the firewall have been exposed, so it seems feasible to add a fourth one, just by adding a new socket factory.

The architecture of the system should change in order to allow Stream connections. In our opinion, this is not a big problem in the architecture, but in the implementation, where the developers chose to only let users use sockets.

For that new protocol, changes have only to be made in the server side. It is the server who establish the connection's features, and later, the client just download the code.

¹In 2.5, the architectural document stays: RMI transport layer normally attempts to open direct sockets to hosts on the Internet, but this attempt is interrupted by the firewall. To bypass the firewall, the transport layer embeds an RMI call within the firewall-trusted HTTP protocol. Firstly, RMI data is sent as the body of an HTTP POST request.

Required technical skills There are not many necessary skills, as we can use an already available implementation of the new protocol. For the testing part, the new communication protocol can be tested easily, with small RMI application. It implies no need for much investment in money neither for special or parallel architecture. Thus this change can add new ways of using RMI without much effort.

3 Changes in the requirements with a big effect on the architecture

In this section we are going to consider two different changes.

- In the first change we consider about making Java RMI interoperable with web services which implement the WSDL and SOAP standards.
- The second change is about the distribution of the remote objects in several servers.

3.1 Interoperability with Web services

Make it possible for a Java program to use a web service using the RMI API.

To achieve this goal it is necessary to make Java RMI interoperable with the web services standards WSDL and SOAP. Web services are very popular today. Many companies offer access to services they offer through a web service interface: A provider for weather forecasts may make his data available for third party applications. The advantage of web services is that there is a big support by libraries for almost every popular programming language.

Need for the change Although there already exist libraries for web services in Java many developers may prefer it not to change an existing program which uses RMI and just to upgrade the RMI library in use to a new version. This problem is even more serious if you do not have access to the source code of two software packages A and B, where A uses web services and B uses Java RMI but still need to find a way to make them interoperable.

Consequences of not making the change Access to certain information or required functionality may be difficult or impossible. To still make this work, you would need to write an adapter, which needs to be maintained. This also implies something about the suitability or usability of RMI, which is something RMI considers to be important.

Risks of making the change Programmer code complexity will be much higher, as it may need to take into account different paradigms and connection modes in a single piece of code.

Difficulties on accommodating semantics from all programming languages to Java and vice versa. As we said RMI really exploits many different features of the Java VM especially on memory management and serialisation of objects as serialized objects obey to the bytecode of Java. Memory management may become hard to implement when RMI still has to support the distributed garbage collection. But if it is not possible developers may need to implement an adapter for every new case where a web service has to be connected to Java RMI. But there is no warrant that the user of the new bridge between RMI and the other system wouldn't have to be very expert in both these systems. The dependency on the Java VM is actually the biggest problem for a realisation of this change. Here we can see best that the philosophies behind RMI and web services are totally different: On the one side RMI wants to be fast and thus the interfaces depend highly on the software for which the interconnection is build for. On the other side web services do not care primarily about efficiency but on compatibility: Interfaces are designed on a business layer and not on a particular implementation.

As [1] showed the performance of Java RMI is really high compared to different systems which are more interoperable. But the performance may decrease when talking to non-RMI systems. This may become a bottleneck even though we can continue to use plain RMI in our current environment.

The last big risk is the greater exposition surface to system attacks, as we have a plethora of serialization and deserialization, which injection attacks may try to use.

Conflicting requirements The extensive use of the Java VM in RMI leads to a great efficiency but after making RMI more interoperable this quality aspect drops.

A great maintainability is archived with the use of transparent object serialisation but if for the reason of interoperability with web services an adapter has to be implemented manually or the client code has to be enriched this quality aspect also drops.

Affected parts If we start to analyse what has to be changed it is obvious that the transport layer has to be exchanged completely: Although both can make use of http-connections the way it is used is totally different. Web services use http just for an envelope of a SOAP message and Java RMI as an envelope for a serialized object. When we have a look what has to be changed to get a message which is serialized in SOAP instead of plain Java bytecode we recognize that the stub has to be completely different. Thus, the way stubs are created has to be changed. This seems to be possible as the architecture of RMI makes extensive use of factories. The architectural description[1] also describes on a high level that a compatibility layer for CORBA has been done. But still even with CORBA there is no full compatibility guaranteed and implementing a serializer for SOAP messages is not a trivial task. The compatibility layer for CORBA is definitively the starting point for a more sophisticated analysis. Details in its implementation may expose more details about RMI's architecture and where changes have to be made or if there could be much reuse of the code of RMI.

Required technical skills To realize a change like this the required skills are very high. Someone skilled both in the RMI internals and in the internal architecture of a web service implementation is needed. The existence of plenty web service libraries like AXIS 2 for Java could become handy, but we think that it easily becomes more a

problem that a help.

In any case, lots of money and time must be spent, and maybe is not the best decision. The implementation of such a large intercommunication architecture does not suit well prototyping, as you need a big stack of things working to get actual results. That is a problem in itself, and can be worse if the testing platform is not controlled by us (e.g. web services that needs payment for their use, or banking systems...).

In any case, if done from a software enterprise, it may benefit lots of people, so we expect to have lots of bug reports, increasing maturity and confidence rapidly.

3.2 RMI Server delegates work to a server farm

*The RMI server transparently offloads any work to a server farm.
The objects used by a single client may reside on different servers in the farm.*

Distributed communication with a central server always has to deal with possible problems with scalability. Because of that, it is very common to use several machines working as only one server. From the client's point of view, he wants to connect to a server, but *this* server could be made up of several computers. This has all the advantages and disadvantages of a distributed approach, such as as increasing the reliability or the performance, in the good side.

Consequences of not making the change In relation to performance, as previously named, one of the reasons to use several machines is distributing the load among them. The server has to manage all the requests from the clients, but when there are a big number of clients connected to it, the workload can become too big. If that is not spread out over several servers, it would be necessary a *strong server*. Having and maintaining such a kind of strong server is more expensive than using a (virtual) server farm (like Amazon EC2).

Besides, regarding reliability, the RMI server is a single point of failure. If the server fails for any reason, your service is interrupted.

Risks of making the change Nowadays, development of parallel algorithm is still a hard task - developers need to take more care if their code works and if it splits up the work efficiently. This also makes development of large system more expensive. The extensive use of relative databases, which already are optimized for parallel computation, makes it easier for developers to split up work on more system, when they operate on tables. Also frameworks like MapReduce decrease development time for operations on key-value pairs, but less has been done for distribution of whole objects.

Another aspect to be taken into account is the Distributed Garbage Collector. The collection must be distributed between several servers, so the leasing algorithm should be modified. But it may still not be adequate: you should know if an object in some other place is not needed, it is not enough with keeping it an amount of time.

Conflicting requirements Currently the data of objects are kept on one server, which

exposes its state through an interface. To enable to have parts of the object graph in several places we come in contact with issues in the field of parallelism like synchronization and concurrency management. This may decrease reliability if there is no good mechanism to restore data if one server fails. If there is a mechanism the efficiency may go down.

Also in current RMI a client connects directly to its server. If we want to split up work on several servers we have two alternatives: It is possible to distribute the objects in use among all servers and run an invoked method only on one of the servers, or to distribute objects only on the servers where the invoked method is executed. The first approach requires a lot of synchronization if data is changed. The last approach on the other hand may lead to a situation where a server A has to ask another server B for a needed object. The server B may still work on a task and server A has to wait. Both situations lower the efficiency if the developer of the system does not care about distribution. If the developer has to think a lot about distribution the goal of RMI to do as much work as possible transparently is not reached. A similar problem exists today in the development of modern many-core-processors with more than 16 cores: Processor architects want their platform be easy to use but also to be efficient.

Affected parts In a way, the RMI server has become a client of the server farm. The skeleton layer should therefore redirect the calls in a different way than usual, which requires changes to the code generated by `rmic`. Details on the change of `rmic` can be found in the discussion of the big trade-off 1.

Distributed garbage collection on the server side also works differently, and may even have to be rethought completely. We also refer here to the discussion of trade-off 1 as the problem is the same.

Required technical skills Much research on how to distribute objects (which may contain code and not only data) has to be done before. Afterwards the required technical skill is still very high. Inspiration could come from the field of high performance computing (HPC): They have to reason about how to divide up work on multiple computers. In the HPC field they make extensive use of the Message passing interface (MPI)². This may become handy as it solves problems in the distribution of tasks.

4 Is the architectural document clear on this?

Can the previous aspects be treated with the information in the architectural document? Or have we needed to search for other information sources?

Small change

From the document, we have extracted the information about the way of RMI actually connects the client to the server, that is, by using sockets. And we could also find that the default use of sockets can be customize by `SocketFactory` in order to introduce concerns about security, authentication... But there has been some missing information

²see <http://www.mcs.anl.gov/research/projects/mpi/>

regarding the use of sockets that would be interesting to include, in order to have a better view of the internals of the system.

In [5], we can see that the `setSocketFactory` method can be called just once. RMI Registry uses the socket factory that is selected by this method, as explained in [3]. Once we contact the registry, the implementation for contacting the elements is downloaded to the client. And we cannot check the different procedures by hand, this can only be done by RMI. However, this is not a problem in the RMI architecture itself, but in its main implementation: this fallback procedure could easily be changed if we had access to the source code of Java RMI. This information is missing in the architectural document and we consider important for a future RMI user to know it.

That also explain another question: can we fallback after starting the connection? It is, if the connection fails, can we retry connecting by another meaning later? It seems not, because it is a RMI's concern to set the way of connection.

Big change 1

With the architectural description we could localize the layers of the RMI architecture which have to be changed. Also it gives the CORBA compatibility layer as a reference for further research. But for a detailed analysis there are still some open questions:

- How is garbage collection been done?

RMI uses reference counting with leases, as mentioned in section 4.3 in the architectural description [1]. However, the paper provides no information on how the distributed garbage collection (DGC) could be customized to archive compatibility with web services. The IBM documentation³ implies that customizing the garbage collection scheme would probably require changes to `DGCClient`. The `DGCClient` itself is not explicitly stated in that article. The DGC lease time is configurable through an API⁴, but this is not officially supported and may not exist in all implementations of RMI.

- Needed stubs for web services and RMI are different. Is it possible to extend the generator that it generates an interoperable stub?

In `rmic`⁵, there is an option for IIOP which is used for CORBA interoperability. Thus it seems to be possible.

- How are callbacks handled and can a server execute code in the client?

In [6] it seems to be possible but only without a firewall⁶

³http://publib.boulder.ibm.com/infocenter/javasdk/v5r0/index.jsp?topic=/com.ibm.java.doc.diagnostics.50/diag/understanding/rmi_dgc.html

⁴<http://download.oracle.com/javase/1.4.2/docs/guide/rmi/sunrmiproperties.html>

⁵<http://download.oracle.com/javase/1.3/docs/tooldocs/win32/rmic.html>

⁶more information at <http://www.cs.swan.ac.uk/~csneal/InternetComputing/ThreadCallBack.htm>

Big change 2

Even if the problem is quite different than the first big change the affected areas in the source code are similar: As the architectural documentation showed this time not the stubs need to be changed but the skeletons. For the generation of the skeletons also `rmic` needs to be modified. This leads to the question whether this can be done easily. As we already mentioned the distributed garbage collection needs to be modified. We localized that this time the affected class might be `DGCServer`. As the affected areas are quite the same as in big change 1 we refer the reader to the documents found there.

5 Conclusions

In general the paper describes in a good way the RMI system, by giving a detailed description of what it is, how it is used and how it can be modified in order to exploit all its capabilities. However, as stated before, we find some lack of information about details in the implementation that force the reader to search for those when he wants to introduce a change. And because of that lack, it is difficult to get a deep view of the system.

References

- [1] Smatanik V., Dérer R., Marek J., Dimitriu A. Java RMI - Software Architecture Document
- [2] Oracle Technology Network. Java Remote Method Invocation - Distributed Computing for Java. <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html>
- [3] Luis-Miguel Alventosa's blog. http://blogs.sun.com/lmalventosa/entry/using_the_ssl_tls_based
- [4] Checklist 1 at the *Course Literature* web page of the course.
- [5] Class `RMISocketFactory` in Oracle, <http://download.oracle.com/javase/1.4.2/docs/api/java/rmi/server/RMISocketFactory.html>
- [6] Comparison to RMI <http://dev.root1.de/wiki/simon> course.