Replication Framework for Jini Services
Reti di Calcolatori LS - a.a. 2003-2004

Jonathan Cristoforetti
160789
jonathan@cristoforetti.name

December 10, 2004

Abstract

Fault-tolerance is a major requirement for software applications that are aimed at certain business sectors, like financial, medical and governmental. A common solution to meet this requirement is the replication of resources, so that in case of a malfunctioning replica, the others can still provide the requested service.

Developing replicated services in a Jini infrastructure, without loosing the features the environment offers, poses some additional challenges. The objective of this article is to discuss some possible solutions and to propose a generic framework for the development of replicated Jini services.

1 Introduction

In several business areas, the importance of data and of services poses some strict requirements on the fault-tolerance of the systems. In some critical sectors, like the financial, the non-availability of a service reaches enormous hourly costs, making fault-tolerance requirements very strict.

One common solution to face these requirements is to introduce replication in the system. Multiple replicas of each resource are made available, so that, in case one of them malfunctions, the service can be provided by the others.

Developing a replicated Jini service in a proper way is not trivial. A quick and dirty solution might lead to the loss of several advantages the Jini environment introduces, like location transparency of the services.

The objective of this article is to discuss some possible solutions to the problem and to provide a framework that will allow for a quick implementation of replicated Jini service, avoiding the need to rewrite code that can be shared among different solutions.

Moreover, the proposed solution will be coherent with the Jini philosophy and will ex-

ploit the environment features to meet the following requirements:

1. Replicated services must be standard Jini services
2. Replication must be transparent to the clients
3. Replicas must be able to sense the current environment and configure themselves accordingly
4. The framework must provide an easy way of creating replicated services from scratch or by adapting existing ones.

2 Replication

Replication of resources can be easily implemented in the case of stateless resources, like printers or idempotent services. However, replicating resources that have an internal state introduces several complications concerning the consistency of the state maintained in the different replicas. Therefore, well defined replica coordination policies are necessary.

There are two main replication models:

- *The passive model*
- *the active model*

2.1 Passive Replication

The passive replication model is the simplest one and is often called Master-Slave in the case of only two replicas. At a certain time, only one instance, the *master*, is active and provides the service, while the others work as a backup. In this model, the master is the only entity visible to the clients and is the only one receiving and processing requests. The only duty of the backup copies is to monitor the master in order to quickly detect problems, and to eventually decide to take over the master.

To address the status update issues, there are two main approaches:

Cold Stand-By: only one instance of the resource is instantiated at a certain time; only in the case of a failure a new instance is created. No information about the state is maintained between the activation of different replicas. This implies a long reconfiguration time and makes this approach better suited for stateless resources, like printers. It must also be noted that the monitoring of the master is done by the support software.

Hot Stand-By: there are at least two instances at a certain time. The master periodically updates the state of the replicas using a checkpoint mechanism. The frequency of the updates is dependant upon the context: more frequent updates mean that the consistency between replicas is maintain for a greater period of time, but it also means greater costs.

2.2 Active Replication

The active replication model dictates that all the instances of the replicated resource are

active at the same time and work together to provide the service. On the one hand, this solves the problem of the state consistency: as each replica updates the state during the execution of the service, it is kept consistent in real-time. On the other hand, having multiple replicas active at the same time introduces new coordination issues, which require very complex solutions to be correctly addressed.

In particular, a service request is usually handled in a five phases process:

1. The request is received
2. The request is distributed to each different replica.
3. The service is performed by each copy, independently or not.
4. The results are collected and compared, a common result is voted, and each replica is validate in order to decide whether it has failed.
5. The result is returned to the client

It can be seen that the sequence of operations is complex and that at least two coordination phases are required. Because of its complexity and its high implementation costs, the active replication is used only in extremely critical situations. For these reasons, this paper is focused only on passive replication.

3 Replication Schema

Designing a passive replication schema means deciding a series of policies regarding the following aspects:

1. Logical organization of the replicas.
2. State synchronization.
3. Failure monitoring.

In the following subsections, each aspect is analyzed.

3.1 Logical Topology

The logical organization of the replicas is an important aspect of a replication schema, as it greatly influences the way in which the replicas coordinate.

A first approach is using a star topology, in which the master directly communicates with all the other replicas present in the group. While simple, this solution is not efficient: the master is loaded with the responsibility of updating the state of all other nodes in the replication group.

A more viable solution is organizing the replicas in a logical list, or chain. The master is the first ring of the chain and directly communicates only with the first slave, which in turn will interact with the following replica and so on (see fig.1).

Assuming the single fault hypothesis holds true, the model can be further simplified. This hypothesis states that one and only one fault can happen at the same time. This means that:

$$TTR = (M)TBF$$

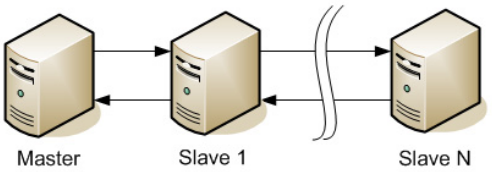


Figure 1: Replicas organized as a logical chain.

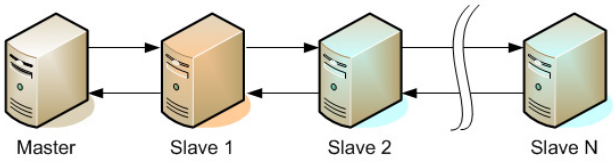


Figure 2: The chain configuration under the single fault hypothesis: the red slave is a hot stand-by replica, while the blue ones are cold stand-by.

where TTR is the Time To Repair and (M)TBF is the (Mean) Time Between Failures. This assumption makes the contemporary presence of more than one slave useless and the model degenerates into a simple Master-Slave.

But if the TTR of the master is not small enough to be less than the TBF, the presence of multiple slaves can still be of use. In fact, one or more extra slaves can be used as cold-stand-by copies and be activated only in the case of a failure. If the time necessary for such activation is lower than the TBF, the single fault hypothesis is satisfied. This leads to the situation depicted in figure 2, where only one slave is hot stand-by and the others are cold. This is the approach that will be used in the framework.

Following the Jini concept of network self-configuration, the replicas should be able to automatically coordinate and organize in the described chain topology. This can be easily achieved exploiting the Jini service lookup system.

Once started, a service replica will first locate the Jini registrar service. Then it will execute a lookup query to determine if another replica is already present and acting as a master for the service.

If no master is currently active, the service replica will configure itself as such, and will register the service by the registrar. If a master is present, the new replica will invoke a join command, asking to join the replication group. If the master doesn't currently have a slave, the join command will succeed and the replica will become the first slave. Instead, if a slave is already present, the master will return its reference. The joining replica will then try the join process on the returned slave, resulting in success or in the return of the proxy corresponding to the next slave in the chain. Figure 3 shows the join protocol.

The process can be visualized as an iteration of join commands on all the chain elements. A recursive solution could also have been used. In that case, upon a join invocation, each element would invoke the join on its successor, wait for the result and return it to the invoker. Even though this solution might be cleaner, it uses resources in the master which can be freed quicker using the iterative solution. In fact, the recursive solution requires the master to wait for n re-

mote invocations, where n is its number of slaves. Adopting the iterative model instead, the master is freed of the need to wait since its the joining replica which must make the remote invocations and wait for the results.

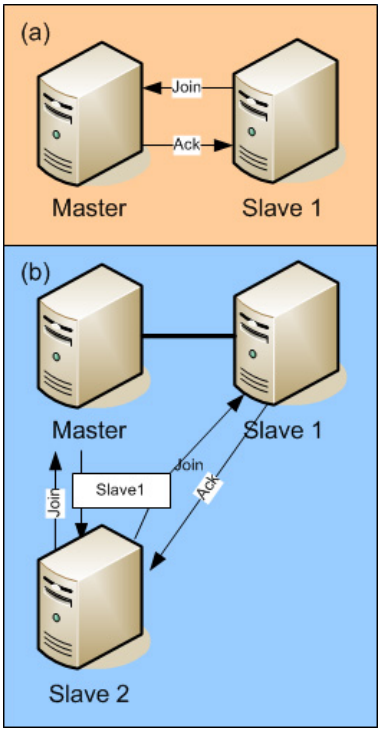


Figure 3: The join protocol: (a) Slave1 joins a Master with no slaves. (b) Slave2 joins Master and Slave1

3.2 State synchronization

Having adopted a passive replication model, a state synchronization protocol must be defined. A checkpoint based protocol can be used for this purpose, in which case the state is synchronized in two situations:

1. When a checkpoint has been reached (i.e. a certain number of request have been served since the previous synchronization).

2. When a replica activates as a first slave and must retrieve a snapshot of the current state from the master

The next step is deciding on what data to send during synchronization. The simplest solution is sending the complete state every time. This might be very bandwidth consuming in situations where the state is complex. A more refined solution is determining the parts of the state which have changed since the last synchronization and send only those. However, keeping in mind the objective of a reusable framework, the first approach seems the most straightforward way to keep the replication support components separate from the service. In fact, if the state can be incapsulated in a serializable object then it can be sent by the replication support

without any knowledge of its internal structure.

In the framework, the state will be synchronized after the requested operation has been executed on the server, but before returning the results to the client. This leads to longer response times, but it is also a safer approach since state will be made consistent between the master and the slave before giving a response to the client.

3.3 Failure detection

In general, detecting a failure of a replication node is a complex problem. A system can show any type of behavior, and determining its correctness requires knowledge about the expected behavior, which is often hard to define. However, making some hypothesis on the possible faults greatly reduces the complexity of the problem. For this reason, in this article replicas are assumed to show a fail-stop behavior in case of a fault, i.e. the replica will halt and the others will be able to detect the failure.

Given this assumptions, an element of the replication chain can monitor its preceding element using a simple heart-beat protocol. Each element of the replication group must periodically invoke a heart-beat method on its preceding element. If the invocation succeeds the replica has not halted, and the fail-stop assumption allows us to assume a replica still functions correctly.

4 Communication protocol between replicas

Let's define the communication interface based on the arguments made in the previous sections.

This interface will be named IReplica and will contain the following operations:

- join: as described in section 3.1, this method is invoked to join the replication group.
- ping: this method is the heart-beat method used to determine failures
- activate: used by the first slave during a takeover of a master, it will activate the second slave to act as the new first slave, i.e. as the new hot stand-by copy.
- synchronize: used to send the updated state to a replica.

5 The replication framework

Focusing on services based on RMI, the classic Jini service registration and usage process can be summarized in the following steps (see figure 4):

1. The service exports a service interface through RMI and registers the Service RMI proxy to the Jini service registrar.
2.

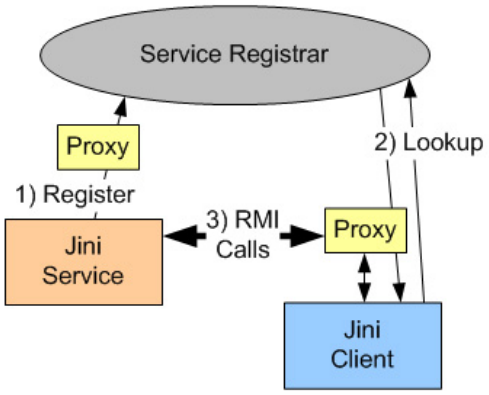


Figure 4: Classic Jini registration and lookup.

3. The client queries the registrar for the service and obtains the Service RMI proxy.

4. The client uses the RMI proxy to invoke the service.

The communication between replicas will also be implemented using RMI, and standard Jini lookup will be used to discover the masters already present in the system.

A first way to do this could be making the master publish a *IReplica proxy* by the Jini service registrar, and making the other replicas use the lookup service to discover it. However this method would prevent the contemporary use of two different services replicated with the framework. Let's show this problem with an example: let Service1 and Service2 be two different services. If a new replica of Service1 executes a query requesting all services which implement *IReplica interface*, the lookup service would return both the Service1 and the Service2 masters, since they both implement IReplica.

The solution is to use a feature of the Java language introduce with Java 1.4: the dynamic proxy. A dynamic proxy is an object which implements a number of interfaces that can be defined at runtime. Using such feature, it's possible to create an object at runtime, which implements both IReplica and the service interface, whichever it will be. The framework will combine the Service proxy and the Replica proxy in a single dynamic proxy and publish it in the Jini lookup service. At startup, replicas will query for services which implement both IReplica and the service interface, and will thus receive the proxies of only the appropriate master. For example, the Service1 proxy would be combined with the IReplica proxy and queries for a service implementing both Service1 and IReplica interfaces would return only the master of Service1. Moreover, a regular client which executes a query for Service1, will receive the dynamic proxy an will be able to use it just like a Service1 proxy. Figure 5 shows the two scenarios.

To create a reusable framework, the interaction protocol between the replication components and the service components must be clearly defined. A solution commonly used in frameworks is the inversion of control pattern: an interface contains a series of callback methods that the framework will invoke during execution, and services that are to be used within the framework must implement such interface.

Given the replication model defined so far, the only interaction needed between the two

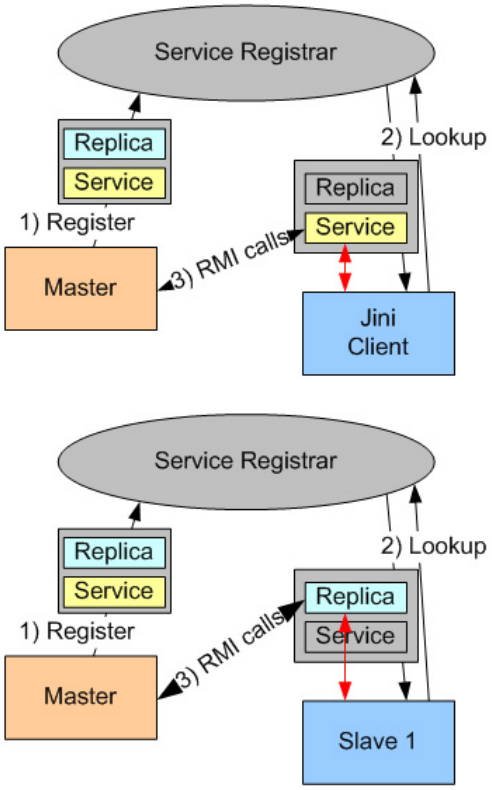


Figure 5: Dynamic proxy in the framework. In the top figure, the client uses the service proxy incapsulated in the dynamic proxy to invoke the service. Instead, in the bottom figure, a node of the replication group uses the replica proxy.

parts is related to the state synchronization, and can be realized using the following two methods:

- getState: this method must return the current state of the replica.
- updateState: this method receives in input an updated version of the state and sets it as the current state of the replica.

With this approach, the framework can be completely independent of the type of service and of the state representation, as long as it is representable by a serializable object. On the other hand, some responsibilities are left on the service programmer regarding the state management. In the case of concurrent requests, for example, the service should freeze all the new incoming requests when the getState method is invoked, and wait for the completion of the requests currently being server before returning the state to the framework. This is necessary in order to guarantee the consistency of the state in the case of concurrent requests.

To summarize, a service, in order to be used with the replication framework, must:

- Be an RMI based remote service.
- Define a service interface.
- Provide a service implementation.
- Implement the callback interface.

Given the fact that the first three requirements are likely to be met by most of the services requiring replication, a programmer

who wants to create a replicated Jini service using the framework must only add an implementation for callback interface.

To easily configure the parameters of the replication group, an XML configuration file is used. In this first version, the configuration file includes:

- The complete name of the service interface
- The complete name of the service implementation
- The frequency of state updates, expresses in the number of service invocations after which to synchronize.

This gives the freedom of modifying the service that must be replicated without the need to recompile the replication framework.

6 Implementation

Figure 6 shows a simplified UML diagram of the framework. The interface and the implementation that a service must provide are indicated as IService and Service respectively. The ServiceProxy class is simply the proxy class created by the JERI export mechanism after the IService interface will be exported. It corresponds to the proxy class that, using JRMP, must be created via *rmic*.

Let's analyze the framework components. The first one, IReplicatedServiceCallback, is the callbacks interface defined in the previous section. As requested, Service implements such interface.

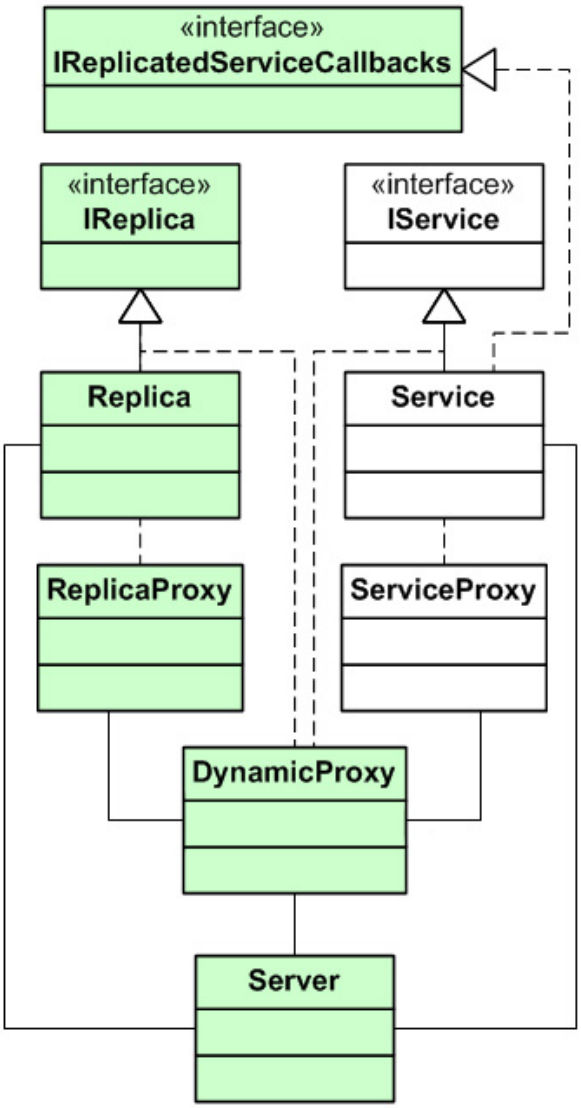


Figure 6: Simplified UML static diagram of the framework

Right below are the interface IReplica and the corresponding implementation class, Replica. The first one defines the communication protocol between the elements of the replication group and is based on the arguments made in section 4. The second implements such interface, incapsulating most of the replication logic. Further down is the class ReplicaProxy, which is nothing more than the proxy class generated by the JERI exporter for the Replica class.

It must be noted that IReplica contains one additional method, called *methodInvoked*, which has been added to allow the object Replica to be notified of each service invocation, so that it's able to determine when the state needs to be synchronized. This was added to the IReplica interface for implementation simplicity, even though it's not logically part of the replicas communication protocol.

The next class is DynamicProxy, which has been described in Section 5. It uses both ReplicaProxy and ServiceProxy to correctly implement both IReplica and IService. It is strongly based on reflection in order to work with any interface a service might have. The only limitation is that the names of the methods in the service interface must not be the same as the methods in the IReplica interface, as they would be erroneously invoked by the framework.

The last class is Server, which has several responsibilities.

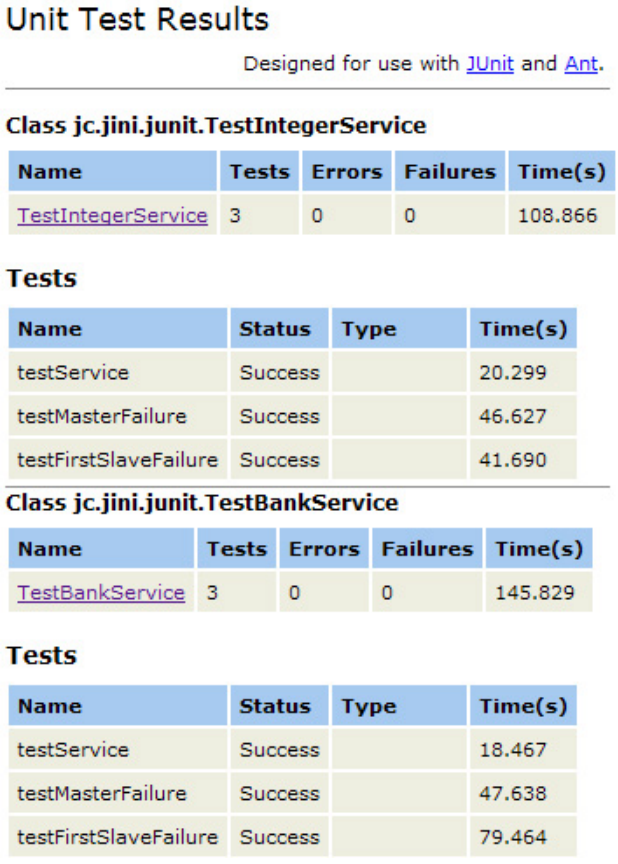
- It must read the configuration files in order to determine the name of the service interface and implementation class.
- It must instantiate the service class and export the relative interface via JERI, obtaining a ServiceProxy object.
- It must instantiate a Replica class and export the IReplica interface via JERI, obtaining a ReplicaProxy object.
- It must combine the two proxies into a DynamicProxy object and register it to all the reggie servers present in the broadcast area.

It's also responsible of dealing with Jini specific tasks, like registration service discovery and lease renewal.

7 Example services and JUnit testing

For the purpose of testing, two example services have been created. The first is a very simple sequential integers generator. The state in this case is an *Integer* object which contains the current integer.

JUnit has been used for testing and for the integer service three tests have been designed. In each one of them, three replicas are started which configure themselves as Master, Hot Slave and Cold Slave. The first test verifies the service under normal condition, where no failure happens. The second one tests the behavior of the replication group when the master replica goes down. The third one analyzes the situation when the Active Slave has a failure. In both cases, the service is continuously invoked a certain amount of times



Unit Test Results

Designed for use with JUnit and Ant.

Class jc.jini.junit.TestIntegerService

Name	Tests	Errors	Failures	Time(s)
TestIntegerService	3	0	0	108.866

Tests

Name	Status	Type	Time(s)
testService	Success		20.299
testMasterFailure	Success		46.627
testFirstSlaveFailure	Success		41.690

Class jc.jini.junit.TestBankService

Name	Tests	Errors	Failures	Time(s)
TestBankService	3	0	0	145.829

Tests

Name	Status	Type	Time(s)
testService	Success		18.467
testMasterFailure	Success		47.638
testFirstSlaveFailure	Success		79.464

Figure 7: JUnit test results

and the replicas are killed and recovered by a separate thread.

The second service is a bank account system, which has been extremely simplified as it is used only for testing. It manages a series of five accounts for which the following operations are defined: getBalance, deposit, withdrawal and transfer. A serializable class called Account is created to model an account and contains an identifier and the current balance. The state of the service is represented by an array of accounts.

As with the integer service, the replication group is tested in three scenarios: no failures, failures of the master and failures of the first slave. In the first case, all operations are tested: money is deposited and withdrawn from all accounts, and money is transferred between them. In the other cases, only the deposit operation is used to modify the state in a way that is easily predictable and verifiable. After some state modifications, the designated replica gets killed and the balance is checked in all accounts to verify the state has properly been replicated. After the replica is recovered, the process is iterated on the remaining replicas.

Figure 7 shows the JUnit report about the tests, which are all successfully passed.

8 Conclusion

Developing replicated Jini services is a complex issue and an error-prone activity, thus designing a reusable framework can prove extremely time-saving in the long term. However, building such framework in a correct

way requires some careful reasoning on the replication policies that will be used. In this article some different options have been discussed and a working replication framework has been build.

Although in a prototypical form, the replication system is reusable, it can be easily customized for different type of services and it meets the proposed goals and requirements. Obviously, there is still a lot of space for improvements. For example, a partial state synchronization system should be developed or the framework could be expanded to support the active replication model.