



Joram 5.4

JNDI

Contents

Contents..... 2

Figures 3

1.JNDI..... 4

1.1.Overview..... 4

1.2.Replication..... 4

 1.2.1.Master ownership strategy..... 4

 1.2.2.Lazy propagation strategy..... 5

 1.2.3.Replicas synchronization..... 5

1.3.Distribution of the naming servers..... 6

1.4.Distribution of the naming contexts..... 7

 1.4.1.Context creation..... 7

 1.4.2.Context name resolution..... 7

1.5.Configuration..... 7

1.6.Loose coupling configuration..... 9

Figures

Figure 1 - JNDI replication.....4

Figure 2 - Lazy propagation.....5

Figure 3 - Replicas synchronization.....5

Figure 4 - Distributed configuration.....6

Figure 5 - Adding a new server.....7

Figure 6 - A 3 servers configuration.....8

Figure 7 - The 3 servers configuration replicas.....9

1. JNDI

1.1. Overview

The goals of the ScalAgent JNDI are to distribute and replicate the naming contexts among various servers in order to provide load balancing and fail over.

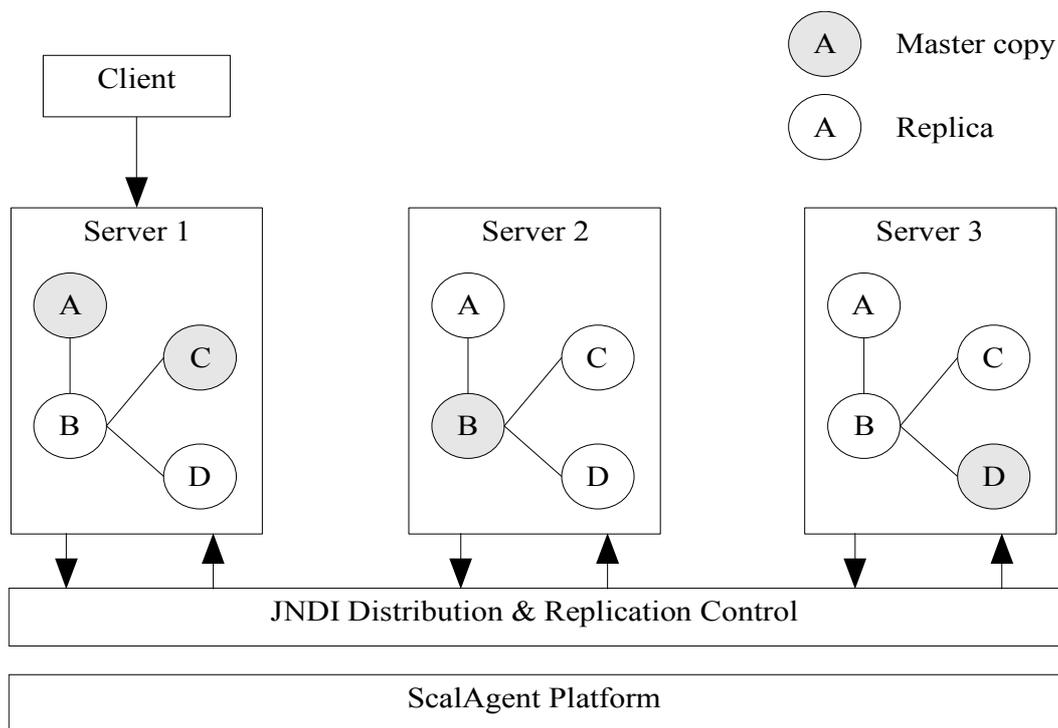


Figure 1 - JNDI replication

The chosen replication follows the master-copy scheme. The master copies are distributed among the servers. The following figure gives an overview of the ScalAgent JNDI architecture. The client sends requests to a server that may communicate with one or more servers of the JNDI configuration. This communication is handled by a control layer responsible for distribution and replication built on top of the ScalAgent platform.

1.2. Replication

1.2.1. Master ownership strategy

The naming contexts are replicated with a master copy replication scheme. Each naming context has one unique master server. Only the master can update the primary copy of the naming context. All other replicas are read-only. Other servers wanting to update the context (bind, rebind, create/destroySubcontext) request the master to do the update.

1.2.2. Lazy propagation strategy

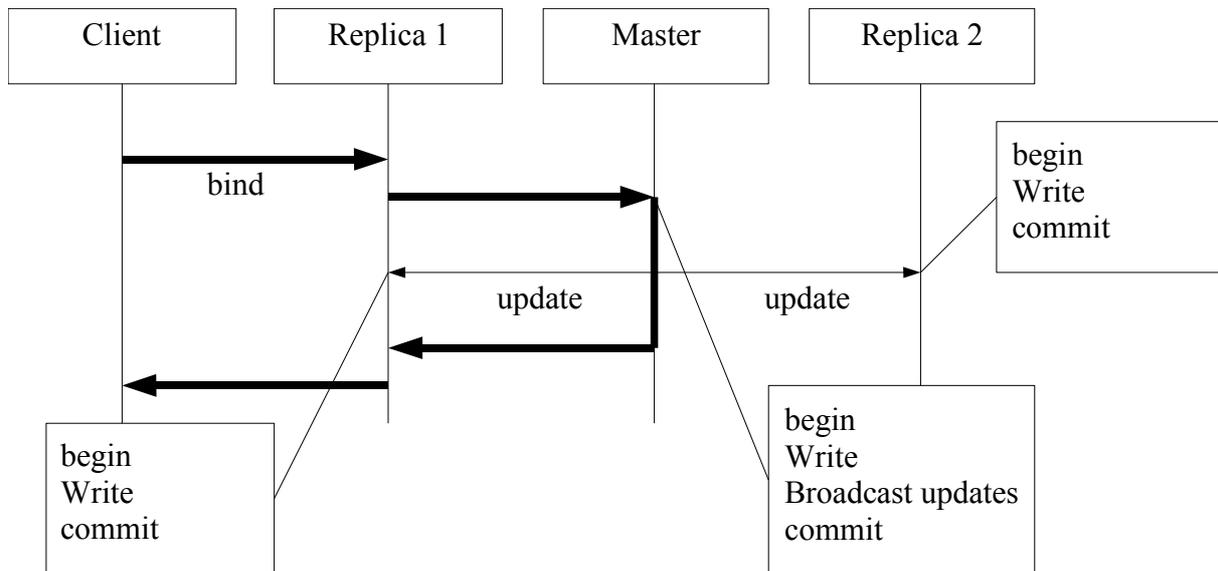


Figure 2 - Lazy propagation

All updates emanate from the master copy of a naming context. The updates are propagated in a lazy way. The master copy is updated in a local transaction that updates the copy and broadcasts the updating notifications to the replicas. Those updates are asynchronously propagated and performed in a separate transaction for each server.

1.2.3. Replicas synchronization

Once a client has done several write requests, he may see different naming data depending on the requested server because updates are asynchronous. The resulting errors are: name not found and stale (out of date) data.

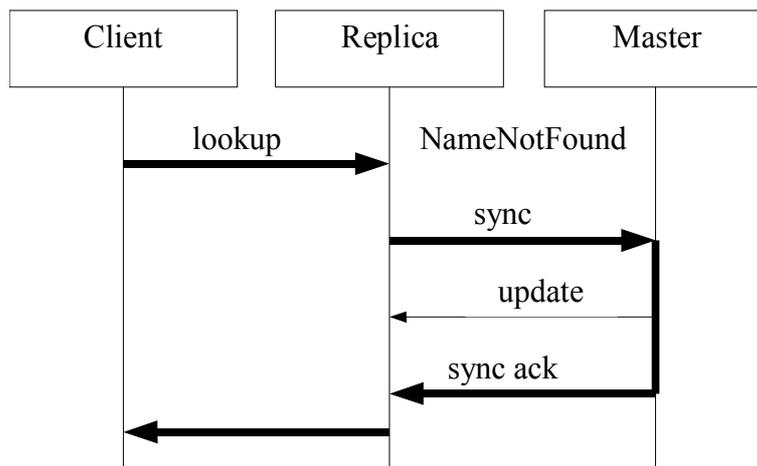


Figure 3 - Replicas synchronization

The name not found error is handled by synchronizing the replica with the master copy of the context. Once synchronized, the request is retried. If the name is still not found, then the client gets the error.

The stale data error cannot be detected. It is inherent to the lazy propagation strategy.

However an explicit synchronization operation for a specific naming context (as described above) could be provided in a JNDI extension interface, enabling the client to get the latest naming data. This extension is still not available.

1.3. Distribution of the naming servers

The naming servers are completely interconnected through the ScalAgent platform. A server owns two entry points: a TCP entry point through which client requests are transmitted and an Agent entry point used to send and receive notifications from the other servers (request forwards, replica updates ...).

The following figure displays a JNDI distributed configuration that includes 4 servers.

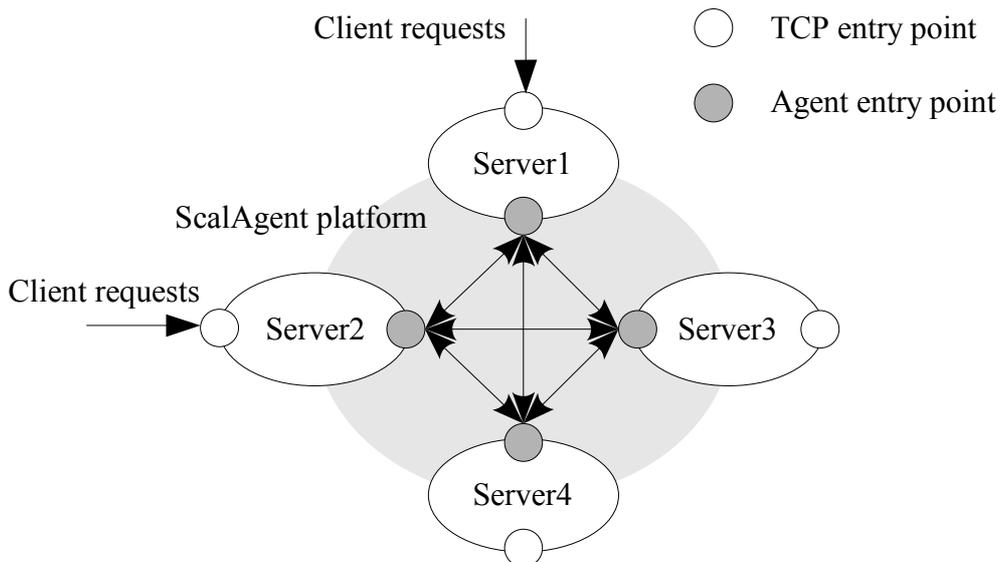


Figure 4 - Distributed configuration

A new naming server is added into a JNDI configuration by assigning it one or more references of other servers already included into the configuration. When a naming server discovers another server, it sends to it an initialization notification containing its master copies and a list of the known servers. In this way a server discovers all the servers of a JNDI configuration.

The following figure displays a sequence diagram of the entrance of a new server into a JNDI configuration. The new server S3 initially knows S1, so it sends to it an initialization notification containing its master copies contexts[S3]. S1 replies with its contexts[S1] and the reference of the server S2. S3 stores the contexts[S1] as replicas, registers the reference of S2 and sends to it an initialization notification.

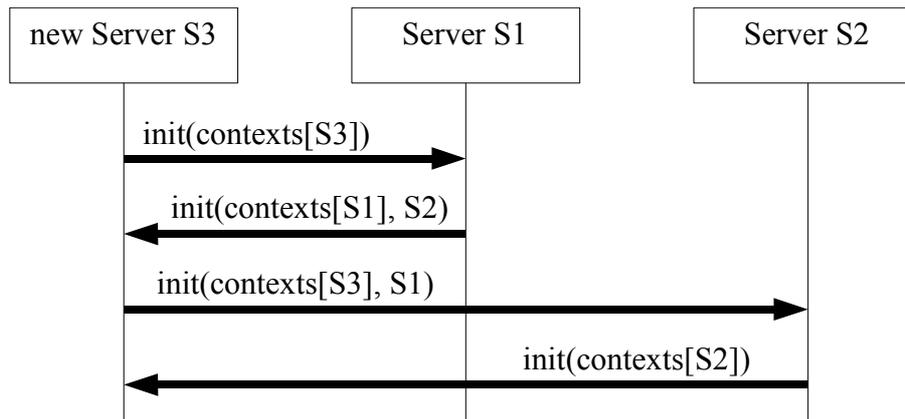


Figure 5 - Adding a new server

A JNDI configuration is wholly replicated. All the servers contain the same (differed) naming data. Some replication groups could be defined in order to reduce the number of updates.

The servers are implemented with the ScalAgent programming model which is asynchronous and reliable. This model is particularly well adapted for the loose coupling of a lazy replication strategy. Moreover this model simply controls the isolation by serializing the operations performed by a server.

1.4. Distribution of the naming contexts

1.4.1. Context creation

When a server (called "local") creates a subcontext of a naming context which master copy is owned by a remote server, it must ask the remote server to create the subcontext. But this subcontext is owned by the server that initiates the creation. So the master copy of this new context belongs to the local server.

1.4.2. Context name resolution

A naming context is accessed by its name which is a path leading from the root naming context to the final name of the context.

Names are not resolved from their path but directly using an index that returns the identifier of the context from its name. In this way a server can handle JNDI requests which path cannot be resolved because of missing intermediate contexts (still not initialized), but which final context locally exists.

This feature is useful if a naming server is started whereas there is a network partition that prevents it from getting the root context. But it can receive some contexts from local surrounding servers and immediately begin to reply to its clients.

1.5. Configuration

A ScalAgent JNDI configuration gathers several JNDI servers. Each server is defined as a ScalAgent service. The class name is *DistributedJndiServer* from the package *fr.dyade.aaa.jndi2.distributed*. The argument line has the following syntax:

```
<listening port> <root owner id> [<server id> ...]
```

- listening port: the port used by the JNDI server to listen to clients requests
- root owner id: the identifier of the JNDI server that owns the root naming context

- server id: the identifier of a JNDI server
 - The listening port and the root owner identifier are mandatory. The list of server identifiers is optional.

A server needs to know the identifier of the root owner in order to compare this identifier with its own identifier. If it is the same, it means that the server is the root owner and must initially create the root naming context. If it is not the same, it means that the server is not the root owner and must register itself with the root owner in order to receive the copies of the naming data owned by the root owner.

The list of JNDI server identifiers is optional. It is useful if the root owner server is not available. In this case the naming data owned by those servers may be accessed (read and write) even if the root server is not available.

The following configuration includes three JNDI servers:

```
<?xml version="1.0"?>
<!DOCTYPE config SYSTEM "a3config.dtd">

<config>
  <domain name="D0" network="fr.dyade.aaa.agent.SingleCnxNetwork"/>
  <server id="0" name="s0" hostname="localhost">
    <network domain="D0" port="27300"/>
    <service class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
args="16400"/>
  </server>
  <server id="1" name="s1" hostname="localhost">
    <network domain="D0" port="27301"/>
    <service class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
args="16401 0"/>
  </server>
  <server id="2" name="s2" hostname="localhost">
    <network domain="D0" port="27302"/>
    <service class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
args="16402 0 1"/>
  </server>
</config>
```

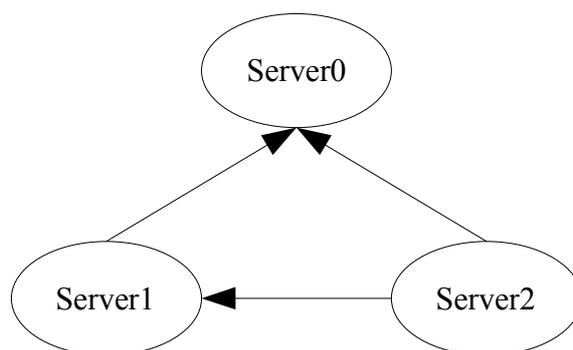


Figure 6 - A 3 servers configuration

The server 0 is the root owner (note that it could have another identifier), the server 1 only knows the root owner server and the server 2 knows both 0 and 1. In this way, if the server 0 is not available when the server 2 starts, it can get the naming data owned by the server 1 anyway.

Let's say that the servers 0 and 1 are first started. The client 0 asks the server 0 to create a subcontext A in the root context. Then the client 1 creates the context B in A. Notice that if the

client 1 creates B before the server 1 received the naming data from the server 0, the request is blocked. When the server 1 receives the copy of A, it retries all the requests waiting for this context (e.g. the creation of B).

If the server 2 is started when the server 0 is unavailable, it can't receive the naming context A. So the client 2 can't do any requests (read and write) in the context A. But the server 2 knows the server 1 so its receives the naming data owned by server 1 (context B). So the client 2 can do a request in B, e.g. create a sub context C.

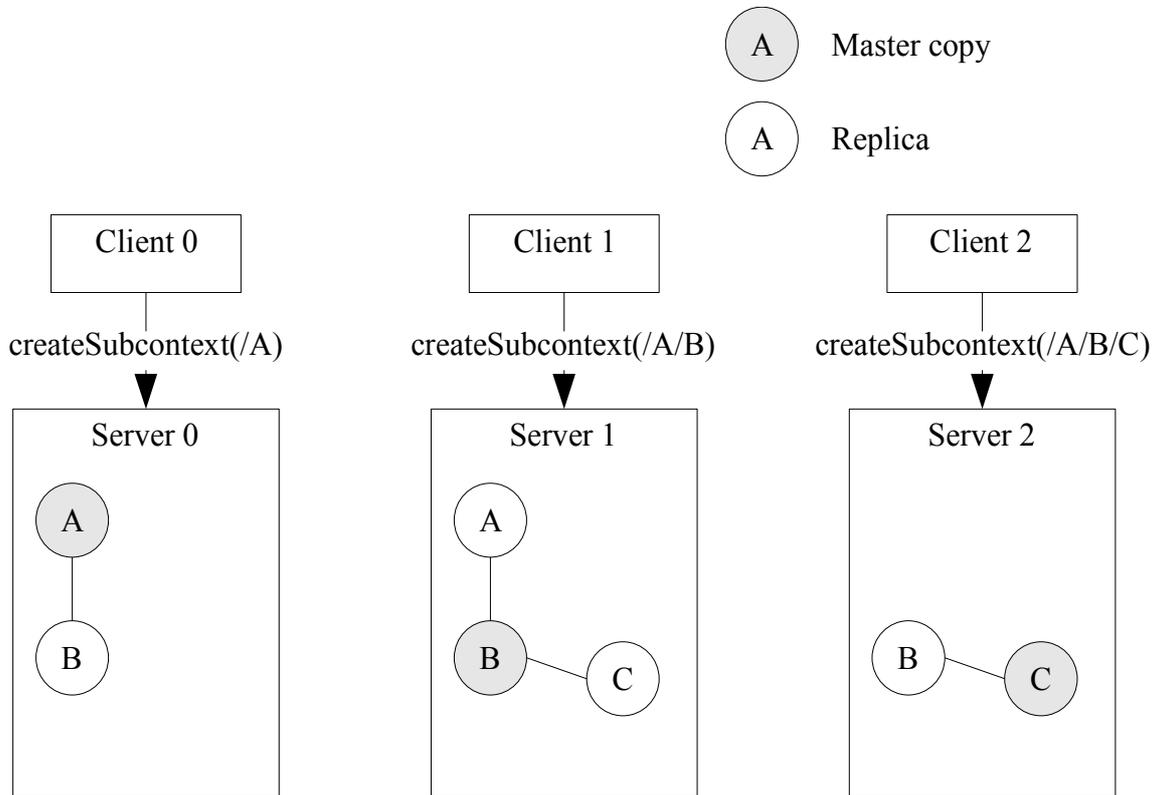


Figure 7 - The 3 servers configuration replicas

The following figure shows the final states of the JNDI servers once the previous scenario has been executed. The server 2 will get the naming data from server 0 as soon as it is available. At this time the server 0 will also get the naming data owned by server 2 (context C).

1.6. Loose coupling configuration

In some cases it can be useful to inhibit the coherency mechanism. The default configuration of the distributed JNDI implies a master / slave behaviour: each JNDI context is controlled by a single server and this server must be alive and reachable in order to allow an insertion or a modification. Setting the `fr.dyade.aaa.jndi2.impl.LooseCoupling` property true allows to inhibit the master / slave behaviour, this property can be set either as a Java environment variable or in the `a3servers.xml` configuration file (see below).

```
<property name="fr.dyade.aaa.jndi2.impl.LooseCoupling" value="true" />
```