

Studienarbeit

FlexiTrust

Using the Shared VM *Jini*^(TM) Utility

by

Dan Dobre (Matrikelnr. 882095)

and

Matthias Mühleisen (Matrikelnr. 863726)

Betreuer: Alexander Wiesmaier

Fachgebiet: Theoretische Informatik

Contents

1	Introduction	1
1.1	Scope of the Term Paper (MM)	1
1.2	Why is shared activation useful? (MM)	1
2	Components Overview (MM)	2
3	The Setup of the Application Environment	2
3.1	Starting httpd (DD)	2
3.2	Setup rmid (DD)	3
3.3	Shared VM Creation (DD)	3
3.4	Setup Reggie (MM)	4
3.5	Setup Norm (MM)	5
4	The registration framework	6
4.1	The <code>ServiceStarter</code> Class (DD)	6
4.2	The <code>ServiceStopper</code> Class (MM)	7
4.3	The <code>jws.admin.ServiceLocator</code> Class (DD)	8
4.4	The <code>ActivationJoinManager</code> Class (MM)	8
4.5	The <code>ActivatableServiceImpl</code> Class and <code>ActivatableServiceRmi</code> (DD)	8
4.6	Changes applied to existing classes (MM)	8
4.7	<code>ServiceStarterTask</code> (DD) and <code>ServiceStopperTask</code> (MM) . .	9
4.8	Additional Tools (DD)	10
5	How to implement a new activatable service by example	10
5.1	The <code>CheckClockImpl</code> service (DD)	10
5.2	Interfaces to implement respectively classes to extend (MM) . . .	10
5.3	Constructors to implement (DD)	10
5.4	Methods to implement (MM)	11
5.5	Methods to overwrite (DD)	12
5.6	Runnig the <code>CheckClockImpl</code> service with ANT (MM)	12

1 Introduction

The initials in the table of contents (i.e. MM resp. DD) stand for the work done by Matthias Mühleisen respectively Dan Dobre on this term paper.

1.1 Scope of the Term Paper (MM)

The scope of this term paper deals with the improvement on performance, reliability, availability and flexibility of the *CA* key components. At the moment, all services within the *CA* plus the isolated `CheckClockImpl` service in the *JWS* framework are none activatable *rmi* remote objects. This means that each of them is residing in it's own virtual machine, a heavy weight process that consumes many computational resources and memory on the machine it's running on. The gain on performance is due to the fact that multiple services that form together a *shared group* are running in one *JVM*^(TM), thus saving a lot of resources. In case of heavy load, the performance of the activatable services (i.e response time) is increased drastically in comparison to the former implementation. The activation framework is more reliable then the traditional *rmi* that came with older versions of *Java*^(TM). When the *rmiid* is (re)started due to a system failure then it instantiates the remote objects automatically from it's log file calling a special constructor on them. This gives you the possibility to load persistent data from a serialization file, thus restoring the state of the object to what it has been before the crash. The automatism regarding to the restarting of services has also an impact on the availability of the application. Since the application is restarted immediately when the machine turns on and the clients don't need to acquire new remote references, there is the guarantee of high availability for all services. Flexibility is given by the fact that the `ServiceStarter` and `ServiceStopper` both have been developed as generic service administration tools that can handle every type of service that implements certain few operations.

1.2 Why is shared activation useful? (MM)

As one example, suppose you have a service in your federation of Jini technology-enabled services and/or devices that needs be accessible all the time. If the service crashes (because of a program error, or because the machine on which it is running crashes), you would like to have it automatically restarted. As another example, to save computational resources, you might want a service to shut down when it has no active clients and restart when a client connects. The RMI activation mechanism, which first appeared in the *Java*^(TM) 2 *SDK, Standard Edition, v 1.2*, is designed to satisfy these requirements. *Jini*^(TM) provides in version 1.2 a framework for starting multiple services in one *Java*^(TM) *Virtual Machine*, a feature which is based upon activation.

2 Components Overview (MM)

The new registration method for activatable services consists of two major components, **the setup of the application environment** and **the registration framework** based upon it.

The registration framework has two tasks: first to register services with the *rmid* so that they run in the same *Java*^(TM) *Virtual Machine*, second to join and unjoin services to and from the lookup service. The registration and deletion can be done either by multicast or unicast.

The new components that fulfill these tasks are `jws.ActivatableService`, `jws.admin.ServiceStarter`, `jws.admin.ServiceStopper`, `jws.admin.ServiceStarterTask`, `jws.admin.ServiceStopperTask`, `jws.admin.ServiceLocator`, `jws.help.ActivationJoinManager`, `jws.impl.ActivatableServiceImpl`, `jws.rmi.ActivatableServiceRmi` which we will describe later in detail.

3 The Setup of the Application Environment

The Application Environment consists of five different components that have to be started in a predefined order¹. The components are *rmid*, *httpd*, *sharedVM*, *reggie* and *norm*. In order to setup successfully the environment you need the following jar packages from *Jini*^(TM) Version 1.2: `jini-core.jar`, `jini-ext.jar`, `create-dl.jar`, `create.jar`, `reggie.jar`, `reggie-dl.jar`, `norm.jar`, `norm-dl.jar`, `tools.jar`, `sharedvm.jar`. The following additional software are needed: *Ant*^(TM) Version 1.5.1, *Java SDK 2*^(TM) Version 1.4.0, *Genjar 1.0.0*

Further notes on the *Shared JVM* and how it is used in the *Jini*^(TM) *Technology Starter Kit* are mentioned in the *Jini*^(TM) Documentation Version 1.2.

Please note that example policy files for the *rmid*, *shared VM*, *norm* and *reggie*, has been included in the policy directory of the *Jini*^(TM) *Technology Starter Kit*; however, you may wish to use a policy file that has been customized for your environment.

3.1 Starting httpd (DD)

Starting the http server:

```
java -jar <jar_dir>/tools.jar -port <port> -dir <jar_dir> -verbose
```

<jar_dir> is the directory in which the `tools.jar` file is located. <port> is the port where the http server is listening. The second <jar_dir> is the http server root. This path should point to the directory where the jar files of the *Jini*^(TM) services reside.

¹see `build.xml` in module `jws`

3.2 Setup rmid (DD)

Starting the rmi daemon (activation system daemon):

```
rmid -J-Djava.security.policy=<security_policy_file_arg>
-log <rmid_log_dir>
```

<security_policy_file_arg> is the absolute path to the security policy file. <rmid_log_dir> specifies the name of the directory the rmi daemon uses to persist information about the activatable services registered with it.

The activation system daemon must be started before activatable objects can be either registered with the activation system or activated in a JVM.

The policy² file used by default for rmid and all the services is `policy.all`:

```
grant {
    permission java.security.AllPermission "", "";
};
```

Stopping the rmi daemon:

```
rmid -stop
```

3.3 Shared VM Creation (DD)

Start the *Shared VM*³:

```
java -jar <jar_dir>/create.jar http://<hostname>:<port>/create-dl.jar
<security_policy_file_arg> <shared_vm_log_directory>
-Djava.security.policy=<shared_vm_policy_file>
```

<jar_dir> is the path that points to the directory where `create.jar` is located.

<hostname> is the name of the host where `create-dl.jar` can be downloaded.

<port> is the port where the http server is listening.

The value of <security_policy_file_arg> will be used as the security policy file for the shared group object. The shared group object is registered with rmid when the shared VM is first created.

<shared_vm_log_directory> This argument is used to specify the log directory that the shared VM will use to persist its state. It will be used by subsequent service creation invocations in order to register services in this particular group. This directory cannot already exist. It will be deleted if the

²For the use of more restrictive policies please see <http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/rmid.html>

³Please see the Shared VM Creation manpage for further details on starting a shared VM.

shared group object's destroy method is invoked through its associated administration interface.

`<shared_vm_policy_file>` The value of this property will be used as the security policy file for the shared VM itself (not to be confused with the shared group object's policy file). Please note that an example policy file for the shared VM has been included in the policy directory of the Jini Technology Starter Kit; however, you may wish to use a policy file that has been customized for your environment.

3.4 Setup Reggie (MM)

Starting reggie⁴ in the *Shared VM*

```
java -jar <jar_dir>/reggie.jar http://<hostname>:<port>/reggie-dl.jar
<security_policy_file_arg> <reggie_log> <jini_group>
<shared_vm_log_directory>
```

`<jar_dir>` is the path that points to the directory where `reggie.jar` is located.

`<security_policy_file_arg>` is used to specify the security policy file that should be passed to the activation group descriptor (`java.rmi.activation.ActivationGroupDesc`) for the lookup service. The value of this property will be used as the security policy file for the server JVM. Please note that an example policy file for Reggie has been included in the policy directory of the Jini Technology Starter Kit; however, you may wish to use a policy file that has been customized for your environment.

`<reggie_log>` is used to specify the absolute path of the log directory that the lookup service should use to persist its state. This directory cannot already exist. It will be deleted if the lookup service's destroy method is invoked through its associated administration interface.

`<jini_group>` An optional argument that specifies a comma-separated list of multicast discovery groups. If provided, it must be the fourth argument passed to the main method of `CreateLookup`. This argument supplies `CreateLookup` with the names of the groups in which the lookup service will advertise that it has membership. If no group list is specified, the Reggie implementation of the lookup service will advertise that it has membership in the "" (or "public") group. The string "public" is interpreted as "". The string "none" indicates that Reggie will initially be a member of no groups, meaning that it will ignore all multicast discovery requests except when the group(s) being requested are null (`ALL_GROUPS`). If a client is configured to discover `ALL_GROUPS`, Reggie will respond and allow those clients to use it. The value of this argument is a single string, consisting of comma-separated group names with no spaces. Multiple groups can be specified by separating each group with a "," (e.g. "ConferenceRoom,FourthFloor,AdvRDGroup"). The setup JVM always starts up Reggie initially as a member of no groups, and then makes a remote

⁴Please see the Reggie - A *Jini*^(TM) Lookup Service manpage for further details.

call through the administrative `DiscoveryAdmin` interface to set the desired member groups; as such, Reggie may briefly advertise itself with multicast announcements containing no groups before subsequently advertising itself with announcements containing the specified groups.

Once Reggie is started, if you wish to have Reggie join other groups, you can do it programmatically. First, retrieve a `JoinAdmin` interface to Reggie. Then, through that `JoinAdmin` instance, invoke the `JoinAdmin.setLookupGroups` method with the desired groups that Reggie should discover and join. If you don't do this, Reggie will actually register with only one lookup service – itself – even though it will advertise itself as being a member of each group specified by the value of the groups argument (or no groups if the groups value was "none").

`<shared_vm_log_directory>` An optional argument that lets the user specify which shared VM will host the service. This directory should exist and should have been created when that shared group was created.

`<hostname>` is the name of the host where `reggie-dl.jar` can be downloaded from.

`<port>` is the port where the http server is listening.

3.5 Setup Norm (MM)

Starting norm⁵ in the *Shared VM*

```
java -jar <jar_dir>/norm.jar http://<hostname>:<port>/norm-dl.jar
<security_policy_file_arg> <norm_log> <jini_group>
<shared_vm_log_directory>
```

`<jar_dir>` is the path that points to the directory where `norm.jar` is located.

`<norm_log>` This argument is used to specify the path to the log directory that the Norm service should use to persist its state. This directory cannot already exist. It will be deleted if the Norm service's `destroy` method is invoked through its associated administration interface.

`<jini_group>` If no groups are contained in this argument, the default action Norm takes is to join lookup services in the "" (or "public") group. The string "public" is interpreted as ""; and the string "none" indicates that Norm should not use multicast (group) discovery to find the lookup services it will join. Multiple groups can be specified by separating each group with a "," (e.g. "com.myCompany.ConferenceRoom,services.FourthFloor,AdvRDGroup"). The string "all" indicates that Norm should attempt to discover and join all lookup services within reach, through multicast discovery.

⁵Please see the Norm - A *Jini*^(TM) Lease Renewal Service manpage for further details.

<hostname> is the name of the host where `norm-dl.jar` can be downloaded from.

<port> is the port where the http server is listening.

4 The registration framework

The registration framework has two tasks: first to register services with the `rmid` so that they run in the same *Java^(TM) Virtual Machine*, second to join and unjoin services to and from the lookup service. The registration and deletion can be done either by multicast or unicast. The components that build up the framework are:

1. The `jws.admin.ServiceStarter` provides static methods for starting a service.
2. The `jws.admin.ServiceStopper` provides static methods for stopping a service.
3. The `jws.admin.ServiceStarterTask` and `jws.admin.ServiceStopperTask` are Ant tasks for starting and stopping a service using the classes mentioned above.
4. The `jws.admin.ServiceLocator` provides a static lookup method for detecting services.
5. The `jws.help.ActivationJoinManager` provides a static method for unicast resp. multicast registration of a service.
6. The `jws.impl.ActivableServiceImpl` and its Interfaces provide common administration methods for every activatable service.

4.1 The ServiceStarter Class (DD)

The `ServiceStarter` registers the desired service with the activation system through a wrapper for activatable services, the jini `ActivateWrapper` object. All services registered using the same `<shared_vm_log_directory>` directory are expected to run in the same *Shared VM*. Synopsis:

```
java -Djava.rmi.server.codebase=<hostname>:<port>/<service>.jar
-cp <service>.jar jws.admin.ServiceStarter
http://<hostname>:<port>/<service>.jar <policy_file>
<service_log_directory> <jini_group> <shared_vm_log_directory>
-serviceName <service_name> -serviceArgs <service_args>
-numOfServices <#services_to_be_started>
-reggieLocation <reggie_location>
-jiniGroups <jini_groups>
```

if `<#services_to_be_started>` is not mentioned, then start one service
if `<reggie_location>` (e.g. localhost) is not mentioned, then do multicast registration

<jini_groups> Comma separated list of jini groups the service should join to when it gets started.

For further details on implementation issues see `jms.admin.ServiceStarter` class.

4.2 The ServiceStopper Class (MM)

The `ServiceStopper` supports a couple of stopping policies and is used to remotely deactivate a service through a lookup to the LUS. Synopsis:

```
java -Djava.rmi.server.codebase=<hostname>:<port>/<service>.jar
-cp <service>.jar -Djava.security.policy=<policy_file>
-Djava.rmi.server.codebase=http://<hostname>:<port>/jini-ext.jar
jws.admin.ServiceStopper -serviceName <service_to_be_stopped>
-numOfServices <#services_of_this_type>
-reggieLocation <reggie_location>
-jiniGroups <jini_groups>
```

<jini_groups> Comma separated list of jini groups as a selection criterion for stopping services. All the stopping mechanisms described below are working with and without specifying the jini group, thus the jini group acts as a constraint.

If <service_to_be_stopped> is not mentioned, then stop all services that implement the `deactivate()` method.

If <#services_to_be_stopped> is not mentioned, then stop all services of given type.

If <reggie_location> (e.g. `localhost`) is not mentioned, then stop services through multicast.

Valid combinations are:

1. `null` (then all services are stopped through multicast)
2. <service_to_be_stopped> Stops all instances of this type through multicast.
3. <reggie_location> Stops all services registered at this specific location.
4. <service_to_be_stopped> <reggie_location> Stops all instances of this type registered at this specific location.
5. <service_to_be_stopped> <reggie_location> <#services_of_this_type> Stops a given number of instances of this type registered at this specific location.

For example⁶ if you want to stop a particular type of service you must mention the location where the service is registered, because stopping through multicast wouldn't make any sense.

⁶Please see `jws/dev/build.xml` for an example service

4.3 The `jws.admin.ServiceLocator` Class (DD)

The `jws.admin.ServiceLocator` represents a kind of lookup helper for services registered with the *LUS*. It provides two static methods `lookup(Class clazz)` and `lookup(Class clazz , String[] jiniGroups)` that return the remote stub or proxy of the desired service. The method uses only multicast for discovery of *LUS*es. It registers a `DiscoveryListener` with the `LookupDiscovery` who's `discovered()` method is called when a *LUS* was found. Until this happens, the main thread waits for synchronization, thus calling the method hangs the client.

For further details on implementation issues see `jms.admin.ServiceLocator` class.

4.4 The `ActivationJoinManager` Class (MM)

The `ActivationJoinManager` is a modified `JoinManager` for activatable services to be registered with *reggie* as `LookupService` and *norm* as `LeaseRenewalService`. The `ActivationJoinManager` provides two different static `join()` methods, for multicast and respectively for unicast. Both of the methods create a new instance of the `ActivationJoinManager` class. When a new instance is created it gets as parameter the remote stub or proxy of the activatable service. This is in general an `ActivatableServiceRmi` object. The *LUS* the service gets registered with, is located either through multicast or unicast. Then a `LeaseRenewalSet` is created between the activatable service and the `LeaseRenewalService`. At this point the `LeaseRenewalService` is updating the lease duration of the activatable service with the *LUS* instead of the service itself. From time to time the `LeaseRenewalService` wakes up the activatable service through a `ExpirationWarningEvent` in order to renew the initial lease of the `LeaseRenewalSet` and to check if the service is still reachable through *rmid*.

For further details on implementation issues see `jms.impl.ActivatableJoinManager` class.

4.5 The `ActivatableServiceImpl` Class and `ActivatableServiceRmi` (DD)

The `ActivatableServiceImpl` class provides the basic implementation of the `java.rmi.ActivatableServiceRmi` interface and *must* be extended by every activatable service. It provides methods that are commonly used by every activatable service and also by the registration framework.

For further details on implementation issues see `jms.impl.ActivatableServiceImpl` class.

4.6 Changes applied to existing classes (MM)

The `CheckClockHelper` class uses the `ServiceLocator` instead of the `JiniHelper` to locate the services. The `ServiceLocator` class can be used to locate the old as well as the new activatable services. `CheckClockImpl` class was modi-

fied appropriately to work in the framework. `CheckClockRmi` does not extend `EquipmentRmi` anymore.

4.7 ServiceStarterTask (DD) and ServiceStopperTask (MM)

`jws.admin.ServiceStarterTask` and `jws.admin.ServiceStopperTask` are wrapper for the `ServiceStarter` respectively `ServiceStopper` classes and represent interfaces to the ant environment. This cut-out of the `build.xml` shows the target declaration:

```
<target name="declare">

    <taskdef name="startService"
            classname="jws.admin.ServiceStarterTask"
            classpath="<jar_dir>/<jar_name>"/>
    <taskdef name="stopService"
            classname="jws.admin.ServiceStopperTask"
            classpath="<jar_dir>/<jar_name>"/>

</target>
```

The following cut-outs of the `build.xml` show how to start and to stop one or more services.

```
<startService

    codebase = "http://<hostname>:<port>/<service_jar>"
    classpath = "<jar_dir>/<service_jar>"
    policyfile = "<jar_dir>/<policy_file>"
    logdir = "<service_log_dir>/service_checkclock_log"
    grouplogdir = "<shared_vm_log_dir>"
    servicename = "<impl_class>"
    serviceargs = "<service_args>"
    numofservices = "<num_of_services_to_be_started>"
    reggielocation = "<hostname>"
    jinigroups = "group1,group2,..."

/>
```

```
<stopService

    codebase = "http://<hostname>:<port>/<service_jar>"
    classpath = "<jar_dir>/<service_jar>"
    policyfile = "<jar_dir>/<policy_file>"
    servicename = "<impl_class>"
    numofservices = "<num_of_services_to_be_stopped>"
    reggielocation = "<hostname>"
    jinigroups = "group1,group2,..."

here. />
```

For further details on implementation issues inspect `jws.admin.ServiceStarterTask` and `jws.admin.ServiceStopperTask` class.

4.8 Additional Tools (DD)

Due to a bug, the actual *Jini*^(TM) distribution does not support multiple class-paths for services using the shared VM utility. This makes obvious the need for a tool that builds the transitive cover of classes needed by the service. To accomplish this task we are using *GenJar*⁷. here. For the exact usage of *GenJar* see `jws/dev/build.xml`.

5 How to implement a new activatable service by example

5.1 The CheckClockImpl service (DD)

Based on the `CheckClockImpl` service we will try to give a step by step example on how to implement and to run a new activatable service.

5.2 Interfaces to implement respectively classes to extend (MM)

The implementation class of the service, in this case `CheckClockImpl`, needs to implement its remote interface `CheckClockRmi` which in turn implements `java.rmi.Remote`. The service needs also to extend `jws.impl.ActivatableServiceImpl`, which provides common administration methods for every activatable service.

5.3 Constructors to implement (DD)

The old constructor is not valid anymore, thus a new constructor is required. The signature is a standard of the activation system. The constructor is called by the activation wrapper (from the *Jini*^(TM) package) when the implementation class gets instantiated due to a call of one of its methods (in lazy mode) and when the activation system restarts after to a computer crash (eager mode). *Jini*^(TM) uses by default eager mode so the services remain started all over the time.

⁷see <http://genjar.sourceforge.net/>

Here you can see the constructor of the `CheckClockImpl` service:

```
public CheckClockImpl(ActivationId id, MarshalledObject data,
    SharedActivation ref) throws RemoteException, IOException,
    ClassNotFoundException {

    super(id, data);
    tryRestoreState();
}
}
```

The arguments are:

1. The `ActivationId` is the key that is used to activate the service in case of a method call in lazy mode or after a system crash in eager mode.
2. The `MarshalledObject` contains the absolute path to the file where the persistent attributes are serialized.
3. The `SharedActivation` is an intermediate between the sharedVM services and the activation runtime but we don't use it here.

In the example above both the constructor of the superclass, and the method `tryRestoreState()` are called. The configuration above is the minimalistic, that means that every activatable service must at least call the method shown above. The `tryRestoreState()` method tries to read the state of the object from the serialization file, and thus restoring the service's state properly.

5.4 Methods to implement (MM)

```
protected void init(String[] args) throws JwsException,
    RemoteException {

    //initialize the objects properties
    //do a snapshot of the properties
    doSaveState();
}
}
```

The method `init(String[] args)` is used in order to pass over command-line arguments to the service. To adapt an existent service to the new registration framework just replace the `main(String[] args)` method by `init(String[] args)`. There is still the need for modifying the `init()` method appropriately.

You shouldn't call `saveState(ObjectOutputStream out)` directly. In order to make a snapshot of the attributes current values just call the `doSaveState()` method, which is an indirection that handles the creation, and closing of the streams.

5.5 Methods to overwrite (DD)

If you want to store and retrieve some additional attributes in your implementation you have to overwrite the following methods:

```
protected void saveState(ObjectOutputStream out) {

    //serializes the attributes of the ActivatioServiceImpl object
    super.saveState(out);
    //use the output stream to write the attributes
    //of the service out to secondary storage

}

protected void restoreState(ObjectInputStream in) {

    //deserializes the attributes of the ActivatioServiceImpl object
    super.restoreState(in);
    //use the output stream to read the attributes
    //of the service from secondary storage into main memory

}
```

For further details on implementation issues see `jms.impl.ActivableServiceImpl` and `jms.impl.CheckClockImpl` classes.

5.6 Runnig the CheckClockImpl service with ANT (MM)

Before starting any activatable service the activation environment must be successfully started. For this purpose the `build.xml` file under `jws/dev/` in CVS module `jws` provides a task called `activationssystem.create`. It will create a log directory and uses the subtarget `activationssystem.start` to launch the environment. You have to start the activation system in a separate window because the `rmid` and `httpd` cannot be started in the background yet, so they will hang. The target `activationssystem.destroy` provides the destruction of the entire activation environment. This includes stopping `httpd`, `rmid` and deleting the persistence log directory for all services. This set of services contains `norm`, `reggie` and all other jini activatable services that are registered with them. You can use a weaker target `activationssystem.stop` to suspend the activation system. This implies only stopping `httpd` and `rmid` without deleting the log directory. The target `activationssystem.start` relaunches the `http` and `rmi` daemon and all the services previously registered with `rmid`.

For starting and stopping the `CheckClockImpl` ant tasks were supplied too. The same `build.xml` file provides the tasks `checkclock.start` and `checkclock.stop`. If you want to test the `CheckClockImpl` run the target `ccTest`. The class hiding behind this task locates the `CheckClockImpl` service and then calls the method `clock()`. You can also use the target `checkclock.test` which launches a check-clock service with `checkclock.start` and afterwards calls the test `ccTest`.

To make all this functionality available the subtargets `httpd.start`, `httpd.stop`, `rmid.start`, `rmid.stop`, `activationssystem.clean`, `reggie.start`, `norm.start`, `sharedVM.start`, `genjar.service`, `genjar.checkclock` are used.

Targets Overview:

1. Environment targets are:

- `activationssystem.start` starts the environment
- `activationssystem.create` deletes all log info and start the environment
- `activationssystem.stop` stops httpd and rmid
- `activationssystem.destroy` deletes all log info and stops httpd and rmid

2. Service targets are:

- `checkclock.start` starts the checkclock
- `checkclock.stop` stops the checkclock
- `checkclock.test` tests the checkclock

3. Other targets used are:

- `declare` declares the `ServiceStarterTask` and `ServiceStopperTask`
- `genjar.checkclock` generates a jarfile containing all classes needed by `CheckClock` to run
- `genjar.service` generates a jarfile containing all classes needed by "service" to run
- `prepare`
- `javac`
- `ajc`
- `jar`
- `compile`
- `rmic`
- `versions`
- `ant.version`
- `aj.version`
- `java.version`
- `ccTest` launches the `TestCheckClock` class
- `httpd.start` and `httpd.stop`
- `rmid.start` and `rmid.stop`
- `sharedVM.start`
- `reggie.start`
- `norm.start`

The sources bundled with this work can be found in the standard cvs repository (/cdc/FlexiPKI/FlexiTrust/cvsRepository). You need the modules "all" and "jws" from the branch "dobredan_2"