Testing LDAP implementations

Emmanuel Lécharny, IKTEK

A LDAP based development cost can be spread over three phases :
- initial analysis
- development
- conformance tests

Each of those phases have more or less the same weight, one third of the total cost. I believe that we can lower the third phase by factoring in the tests globally during the development phase.

As developers must test their code, it's a good idea to use those unit tests for conformance check.

My understanding is that doing so can lower the global cost by one sixth, and reduce the cost of the third phase by more than 50%.

Testing an LDAP application is also a matter of checking its performances, according to the expected requests. It should be possible to run performances tests at different phases in the development, in order to gather the different results and check that they are in line with the expectations.

Testing for performances issues is about checking the different kinds of items:
- request size (check the logs, check the requests)
- request duration
- index setting
- network latencies
- operation sequences (do you need to always bind/op/unbind ?)

We can use some dedicated tools to code those tests - we will provide an overview of three different tools that can be used.

I Jmeter

Jmeter was initially created to record some HTTP tests, but has since its inception extended to allow users to define some LDAP tests.

The good point about Jmeter is that it has a user-friendly UI, and it generates some scripts that can be used offline, with agents running on different computers. To some respect, it's similar to SLAMD, except that it's not a dedicated tool.

Scenarii are way easier to create, as you don't need to code anything: you just add new requests, and then configure them.

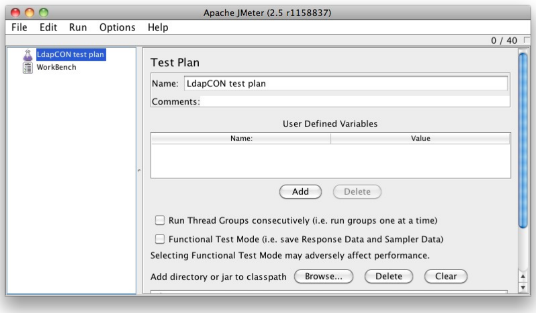
1) Simple LDAP scenario

We will define a very simple test where we will bind on a server, add an entry, do a search, do a delete and unbind.

We first create a test plan, as exposed in the following snapshot.

Then, we have to add a ThreadGroup - to set the number of injectors we will use to run the scenario.

We can start with 20 threads (20 LDAP sessions), and setting the number of loops to 1000 for instance.


Apache JMeter (2.5 r1158837)

Test Plan

Name: LdapCON test plan

Comments:

User Defined Variables

Name:	Value

Run Thread Groups consecutively (i.e. run groups one at a time) [ ]

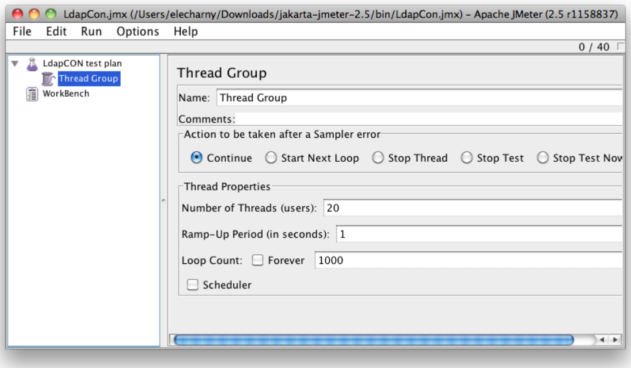
Functional Test Mode (i.e. save Response Data and Sampler Data) [ ]

Selecting Functional Test Mode may adversely affect performance.

Add directory or jar to classpath

Test plan creation

The next snapshot shows the configuration of this Thread Group.


LdapCon.jmx (/Users/elecharny/Downloads/jakarta-jmeter-2.5/bin/LdapCon.jmx) - Apache JMeter (2.5 r1158837)

Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error: [✓ Continue] Start Next Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 20

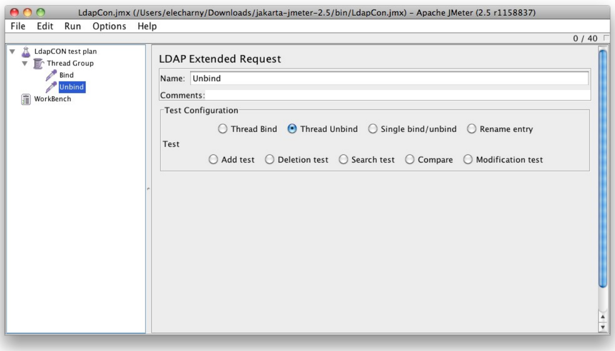
Ramp-Up Period (in seconds): 1

Loop Count: Forever [ ] 1000

Scheduler [ ]

Thread group creation

Now, we can add a Bind request, which is done by adding a LdapExtended request Sampler. The following picture shows what is the configuration for this initial bind request. Each thread will execute this request first. Let's also add a unbind request to close the session:


LdapCon.jmx (/Users/elecharny/Downloads/jakarta-jmeter-2.5/bin/LdapCon.jmx) - Apache JMeter (2.5 r1158837)

LDAP Extended Request

Name: Unbind

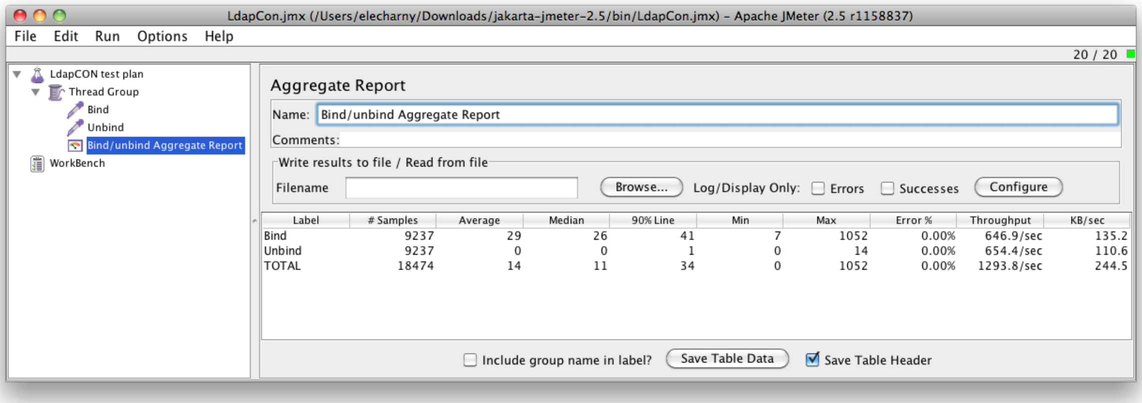
Comments:

Test Configuration

Test: Thread Bind [✓ Thread Unbind] Single bind/unbind Rename entry Add test Deletion test Search test Compare Modification test

Unbind Request

We can add a Listener to expose the results of a running test, and when we run the test, we obtain the following result:


LdapCon.jmx (/Users/elecharny/Downloads/jakarta-jmeter-2.5/bin/LdapCon.jmx) – Apache JMeter (2.5 r1158837)

Aggregate Report

Name: Bind/unbind Aggregate Report

Comments:

Write results to file / Read from file

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
Bind	9237	29	26	41	7	1052	0.00%	646.9/sec	135.2
Unbind	9237	0	0	1	0	14	0.00%	654.4/sec	110.6
TOTAL	18474	14	11	34	0	1052	0.00%	1293.8/sec	244.5

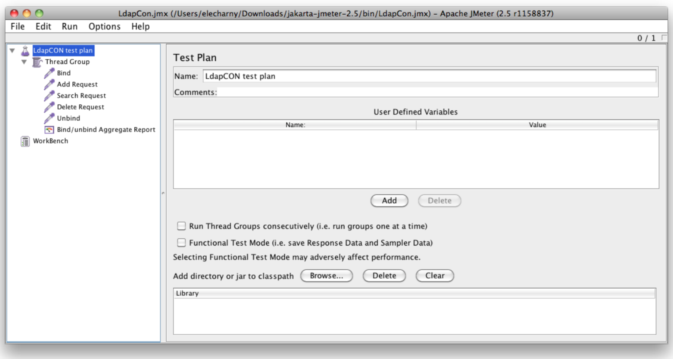
Bind/Unbind results

As we can see, we obtain a throughput for each operation, plus the delay to execute each operation.

Now, to show how easy it is to modify the test, we will add an Add, a Search and a Delete requests.

As we can see, we had to use the thread number in order to avoid collisions when creating entries from different threads. This is done by using the predefined variable __threadNum.

We will use this variable in the following requests. Similarly for search and deletion: we use the __threadNum variable to avoid collisions.


LdapCon.jmx (/Users/elecharny/Downloads/jakarta-jmeter-2.5/bin/LdapCon.jmx) – Apache JMeter (2.5 r1158837)

Test Plan

Name: LdapCON test plan

Comments:

User Defined Variables

Name:	Value

Run Thread Groups consecutively (i.e. run groups one at a time)

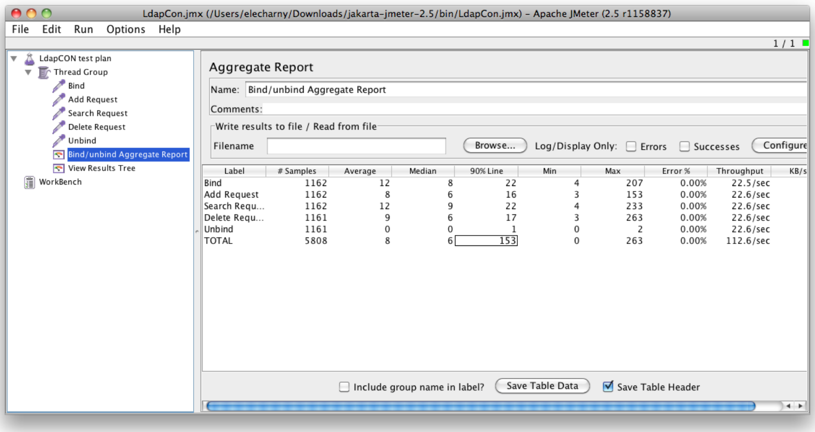
Functional Test Mode (i.e. save Response Data and Sampler Data)

Selecting Functional Test Mode may adversely affect performance.

Add directory or jar to classpath

Add Request

And when we run the test, we obtain the following result:


LdapCon.jmx (/Users/elecharny/Downloads/jakarta-jmeter-2.5/bin/LdapCon.jmx) – Apache JMeter (2.5 r1158837)

Aggregate Report

Name: Bind/unbind Aggregate Report

Comments:

Write results to file / Read from file

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/s
Bind	1162	12	8	22	4	207	0.00%	22.5/sec	
Add Request	1162	8	6	16	3	153	0.00%	22.6/sec	
Search Requ...	1162	12	9	22	4	233	0.00%	22.6/sec	
Delete Requ...	1161	9	6	17	3	263	0.00%	22.6/sec	
Unbind	1161	0	0	1	0	2	0.00%	22.6/sec	
TOTAL	5808	8	6	153	0	263	0.00%	112.6/sec	

Results

II Java Unit testing

We will now go a bit further, exploring how we can test an LDAP Application written in Java using JUnit and Apache DirectoryServer

1) Basic unit test

The key is to define an LDAP Server before starting the test. We have created some Java annotations to do that.

We first extend the AbstractLdapTestUnit class, which defines some common fields, and then it's all about getting a connection and executing some operation on it.

Here is an example:

```
@RunWith(FrameworkRunner.class)
@CreateDS(allowAnonAccess = true, name = "TestLdapServer")
@CreateLdapServer(
    transports =
    {
        @CreateTransport(protocol = "LDAP")
    })
public class SimpleServerTest extends AbstractLdapTestUnit
{

    /**
     * A simple test
     */
    @Test
    public void test() throws Exception
    {
        // Get an admin connection on the defined server
        LdapConnection connection = getWiredConnection( getLdapServer(), "uid=admin,ou=system",
"secret" );

        // Check that we can read an entry
        assertNotNull( connection.lookup( "ou=system" ) );

        // And close the connection
        connection.close();
    }
}
```

2) Improving the test with injection of entries

We may need to have a started server with some base entries already present. This is very easy to do using the @ApplyLdifs annotation.

This lets you inject any consistent entry (assuming that the local schema accepts it). In the following example, we add a new entry which we will authenticate against. The remaining definition of the server is exactly the same.

```
@RunWith(FrameworkRunner.class)
@ApplyLdifs(                                              // Inject an entry
    {
        // Entry # 1
        "dn: uid=elecharny,ou=users,ou=system",
        "objectClass: uidObject",
        "objectClass: person",
        "objectClass: top",
        "uid: elecharny",
        "cn: Emmanuel Lécharny",
        "sn: lecharny",
        "userPassword: emmanuel"
    })
@CreateDS(allowAnonAccess = true, name = "TestLdapServer")  // Define the DirectoryService
@CreateLdapServer(                                          // Define the LDAP protocol layer
    transports =
    {
        @CreateTransport(protocol = "LDAP")
    })
public class LoadedServerTest extends AbstractLdapTestUnit
{

    /**
     * A test where we bind using the added entry credentials
     */
    @Test
    public void testBindUser() throws Exception
    {
        // Get a connection (not bound yet) on the server
        LdapConnection connection = getWiredConnection( getLdapServer() );

        connection.bind( "uid=elecharny,ou=users,ou=system", "emmanuel" );

        // Check that we can read an entry
        assertNotNull( connection.lookup( "uid=elecharny,ou=users,ou=system" ) );

        // And close the connection
        connection.close();
    }
}
```

Note that we can inject as many entries as we need, each new entry starting with a 'dn:xxx' line. Also note that the entries order is significant, as they will be injected from top to bottom.

We can also inject LDIF files, instead of injecting a bunch of ldif entries in the test file. It's just a matter of using the @ApplyLdifFiles annotation instead, as demonstrated in the following example:

```
@RunWith(FrameworkRunner.class)
@ApplyLdifFiles(                                                // Inject an entry
    {
        "User1.ldif",
        "User2.ldif"
    })
@CreateDS(allowAnonAccess = true, name = "TestLdapServer")  // Define the DirectoryService
@CreateLdapServer(                                          // Define the LDAP protocol layer
    transports =
    {
        @CreateTransport(protocol = "LDAP")
    })
public class LdifFileServerTest extends AbstractLdapTestUnit
{

    /**
     * A test where we bind using the added entry credentials
     */
    @Test
    public void testBindUser() throws Exception
    {
        // Get a connection (not bound yet) on the server
        LdapConnection connection = getWiredConnection( getLdapServer() );

        connection.bind( "uid=elecharny,ou=users,ou=system", "emmanuel" );

        // Check that we can read an entry
        assertNotNull( connection.lookup( "uid=elecharny,ou=users,ou=system" ) );

        // and the second one
        assertNotNull( connection.lookup( "uid=pamarcelot,ou=users,ou=system" ) );

        // And close the connection
        connection.close();
    }
}
```

Here we are reading two LDIF files:

```
User1.ldif :
dn: uid=elecharny,ou=users,ou=system
objectClass: uidObject
objectClass: person
objectClass: top
uid: elecharny
cn: Emmanuel Lécharny
sn: lecharny
userPassword: emmanuel
```

```
User2.ldif :
dn: uid=pamarcelot,ou=users,ou=system
objectClass: uidObject
objectClass: person
objectClass: top
uid: pamarcelot
cn: Pierre-Arnaud Marcelot
sn: marcelot
userPassword: pierre-arnaud
```

3) Adding a partition

We can now go a step further: by adding a Partition, we allow a user to define a NamingContext, with some dedicated indexes. We can also create more than one partition.

Those partitions will exist as long as the tests will run, then will be deleted.

Here is an example where we define a partition with three indexes.

```
@RunWith(FrameworkRunner.class)                           // Define the DirectoryService
@CreateDS(
    name = "TestLdapServer",
    allowAnonAccess = true,
    partitions =
    {
        @CreatePartition(
            name = "example",
            suffix = "dc=example,dc=com",
            contextEntry = @ContextEntry(
                entryLdif =
                    "dn: dc=example,dc=com\n" +
                    "dc: example\n" +
                    "objectClass: top\n" +
                    "objectClass: domain\n\n" ),
            indexes =
            {
                @CreateIndex( attribute = "objectClass" ),
                @CreateIndex( attribute = "dc" ),
                @CreateIndex( attribute = "ou" )
            } )
    })
@CreateLdapServer(                                          // Define the LDAP protocol layer
    transports =
    {
        @CreateTransport(protocol = "LDAP")
    })
public class ServerWithPartitionTest extends AbstractLdapTestUnit
{

    /**
     * A simple test
     */
    @Test
    public void test() throws Exception
    {
        // Get an admin connection on the defined server
        LdapConnection connection = getWiredConnection( getLdapServer(), "uid=admin,ou=system",
"secret" );

        // Check that we can read the Example context entry
        assertNotNull( connection.lookup( "dc=example,dc=com" ) );
        // And close the connection
        connection.close();
    }
}
```

4) Saving some execution time

Every time we create a server, run a test, and stop it, we pay a price. Typically, starting the server costs around a second. If we do that for every test, the price will quickly be prohibitive.

Now, let's assume we can use one single server for a set of tests: we will just start the server once, run all the tests, and stop the server. If we have 100 tests, we save 99 seconds. In Apache Directory Server, we have around 1000 tests, we can save as much as 20 minutes while running them.

So far, the way we defined the tests in the previous paragraphs allows us to add more then one test per class.

Are we good then ? Not completely: what about data that might be created by a test, and which can impact another test? Typically, one test may create an entry without deleting it at the end, and the next test may also be needed to create this same entry: we will have some error, as the entry already exists. We can avoid such a situation by carefully reverting all the modifications we made on each test, but this would be a very dull task.

We have a better solution: ApacheDS provides a ChangeLog facility that stores all the modifications done on the server, and the test framework uses this log to automatically revert the operations, computing the reverse operation for each modification.

For instance, if you add an entry, then the system will delete it.

It works pretty well for almost all the operations except Aliases. It's also activated by default.

Let's see how it works with two conflicting tests. At first, we will disable the ChangeLog system

```
@RunWith(FrameworkRunner.class)                             // Define the DirectoryService
@CreateDS(
    name = "TestLdapServer",
    allowAnonAccess = true,
    enableChangeLog = false                                 // Disable the changeLog tracking
    )
@CreateLdapServer(                                          // Define the LDAP protocol layer
    transports =
    {
        @CreateTransport(protocol = "LDAP")
    })
public class ServerWithRevertTest extends AbstractLdapTestUnit
{
    /**
     * A first test
     */
    @Test
    public void test1() throws Exception
    {
        // Get an admin connection on the defined server
        LdapConnection connection = getWiredConnection( getLdapServer(), "uid=admin,ou=system",
"secret" );

        // Create a new entry
        connection.add( new DefaultEntry(
            "cn=test,ou=system",
            "objectClass: top",
            "objectClass: person",
            "cn: test",
            "sn: Test" ) );

        // Check that we can read an entry
        assertNotNull( connection.lookup( "cn=test,ou=system" ) );

        // And close the connection
        connection.close();
    }


    /**
     * A second test
     */
    @Test
    public void test2() throws Exception
    {
        // Get an admin connection on the defined server
        LdapConnection connection = getWiredConnection( getLdapServer(), "uid=admin,ou=system",
"secret" );

        // Create a new entry
        connection.add( new DefaultEntry(
            "cn=test,ou=system",
            "objectClass: top",
            "objectClass: person",
            "cn: test",
            "sn: Test" ) );

        // Check that we can read an entry
        assertNotNull( connection.lookup( "cn=test,ou=system" ) );

        // And close the connection
        connection.close();
    }
}
```

Running those two tests, you will get this exception:

```
org.apache.directory.shared.ldap.model.exception.LdapEntryAlreadyExistsException:
ENTRY_ALREADY_EXISTS: failed for MessageType : ADD_REQUEST
Message ID : 2
    Add Request :
Entry
    dn[n]: cn=test,ou=system
    objectclass: top
    objectclass: person
    sn: Test
    cn: test
: ERR_250_ENTRY_ALREADY_EXISTS cn=test,ou=system already exists!
        at
org.apache.directory.shared.ldap.model.message.ResultCodeEnum.processResponse(ResultCodeEnum.java:2035
)
        at
org.apache.directory.ldap.client.api.LdapNetworkConnection.add(LdapNetworkConnection.java:698)
…
```

If you enable the ChangeLog system, the tests will pass. Just change the parameter in @CreateDS:

```
@CreateDS(
    name = "TestLdapServer",
    allowAnonAccess = true,
    enableChangeLog = true                                  // Allow changeLog to track modifications
    )
```

You won't have errors (but remember that this parameter defaults to true).

Note that every entries added with @ApplyLdifs or @ApplyLdifFiles will be reverted for every tests too. Keep in mind that those modifications are costly operations, so if you have hundreds of entries to inject, the cost of reversing them might be more expensive than not reversing them at all…

5) Ldap server scope

Sometimes, you may need to define some general LDAP server for all the tests, but a few of them might need some specific configuration. The way JUnit is designed allows you to aggregate tests in a class, and to aggregate classes in a test suite. We follow this hierarchy by allowing the user to define a Ldap Server in the suite - and it will be valid for all the tests -, or in a single class - and it will then override the suite server - or even for a single test.

First, let's see how we can define a server for a suite. No surprise, we still use the same annotations:

```
@RunWith ( FrameworkSuite.class )
@Suite.SuiteClasses ( {
        // kerberos
        KeyDerivationServiceIT.class,
        PasswordPolicyServiceIT.class,

        // operations.add
        AddingEntriesWithSpecialCharactersInRDNIT.class,
        AddIT.class,

        // operations.bind
        BindIT.class,
        ...
        } )
@CreateDS(
    name = "SuiteDS",
    partitions =
    {
        @CreatePartition(
            name = "example",
            suffix = "dc=example,dc=com",
            contextEntry = @ContextEntry(
                entryLdif =
                    "dn: dc=example,dc=com\n" +
                    "dc: example\n" +
                    "objectClass: top\n" +
                    "objectClass: domain\n\n" ),
            indexes =
            {
                @CreateIndex( attribute = "objectClass" ),
                @CreateIndex( attribute = "dc" ),
                @CreateIndex( attribute = "ou" )
            } )
    } )
@CreateLdapServer (
    transports =
    {
        @CreateTransport( protocol = "LDAP" ),
        @CreateTransport( protocol = "LDAPS" )
    })
public class StockServerISuite
{
}
```

Here, we have defined a partition, and initialized two different transports, because we want to test LDAPS.

We could have injected some entries, but it's not necessarily a need at this point. The only added entry is the contact entry.

Now, let see a complete example where we define may different LDAP server at different level:

```
@CreateDS( name = "classDS" )
public class DirectoryServiceAnnotationTest
{
    @Test
    public void testCreateDS() throws Exception
    {
        DirectoryService service = DSAnnotationProcessor.getDirectoryService();

        assertTrue( service.isStarted() );
        assertEquals( "classDS", service.getInstanceId() );

        service.shutdown();
        FileUtils.deleteDirectory( service.getInstanceLayout().getInstanceDirectory() );
    }


    @Test
    @CreateDS( name = "methodDS" )
    public void testCreateMethodDS() throws Exception
    {
        DirectoryService service = DSAnnotationProcessor.getDirectoryService();

        assertTrue( service.isStarted() );
        assertEquals( "methodDS", service.getInstanceId() );

        service.shutdown();
        FileUtils.deleteDirectory( service.getInstanceLayout().getInstanceDirectory() );
    }


    @Test
    @CreateDS(
        name = "MethodDSWithPartition",
        partitions =
        {
            @CreatePartition(
                name = "example",
                suffix = "dc=example,dc=com",
                contextEntry = @ContextEntry(
                    entryLdif =
                        "dn: dc=example,dc=com\n" +
                        "dc: example\n" +
                        "objectClass: top\n" +
                        "objectClass: domain\n\n" ),
                indexes =
                {
                    @CreateIndex( attribute = "objectClass" ),
                    @CreateIndex( attribute = "dc" ),
                    @CreateIndex( attribute = "ou" )
                } )
        } )
    public void testCreateMethodDSWithPartition() throws Exception
    {
        DirectoryService service = DSAnnotationProcessor.getDirectoryService();

        assertTrue( service.isStarted() );
        assertEquals( "MethodDSWithPartition", service.getInstanceId() );
        assertEquals( 2, service.getPartitions().size() );

        service.shutdown();
        FileUtils.deleteDirectory( service.getInstanceLayout().getInstanceDirectory() );
    }


    @Test
    @CreateDS(
        name = "MethodDSWithPartitionAndServer",
        partitions =
        {
            @CreatePartition(
                name = "example",
                suffix = "dc=example,dc=com",
                contextEntry = @ContextEntry(
                    entryLdif =
                        "dn: dc=example,dc=com\n" +
                        "dc: example\n" +
                        "objectClass: top\n" +
                        "objectClass: domain\n\n" )
                )
        } )
    @CreateLdapServer (
        transports =
        {
            @CreateTransport( protocol = "LDAP" ),
            @CreateTransport( protocol = "LDAPS" )
        })
    public void testCreateLdapServer() throws Exception
    {
        // First, get the service
        DirectoryService service = DSAnnotationProcessor.getDirectoryService();

        // Check that the service is running
        assertTrue( service.isStarted() );
        assertEquals( "MethodDSWithPartitionAndServer", service.getInstanceId() );
        assertEquals( 2, service.getPartitions().size() );

        assertTrue( service.getAdminSession().exists( new Dn( "dc=example,dc=com" ) ) );

        // Now, get the server
        LdapServer ldapServer = ServerAnnotationProcessor.getLdapServer( service );

        // Check that the server is running
        assertTrue( ldapServer.isStarted() );

        ldapServer.stop();
        service.shutdown();

        FileUtils.deleteDirectory( service.getInstanceLayout().getInstanceDirectory() );
    }
}
```

Note that the Ldap protocol network layer may be different, and one can define a specific port for a specific server. However, those ports are automatically selected not to conflict with any existing port.

6) Injecting Schema elements

Adding some new Schema elements is as easy as injecting the associated LDIF entries. This example demonstrates the injection of an AttributeType and an ObjectClass (you only need to know where in your server the SubschemaSubentry is stored)

```
    @Test
    @CreateDS( name="SchemaAddAT-test" )
    @ApplyLdifs(
        {
            // Inject an AT
            "dn: cn=schema",
            "changetype: modify",
            "add: attributeTypes",
            "attributeTypes: ( 1.3.6.1.4.1.65536.0.4.3.2.1 NAME 'templateData' DESC 'template data'
SYNTAX 1.3.6.1.4.1.1466.115.121.1.5 SINGLE-VALUE X-SCHEMA 'other' )",
            "-",

            // Inject an OC
            "dn: cn=schema",
            "changetype: modify",
            "add: objectClasses",
            "objectClasses: ( 1.3.6.1.4.1.65536.0.4.3.2.2 NAME 'templateObject' DESC 'test OC' SUP top
STRUCTURAL MUST ( templateData $ cn ) X-SCHEMA 'other' )",
            "-"
        }
        )
    public void testAddAttributeTypeObjectClassSubSchemaSubEntry() throws Exception
    {
        checkAttributeTypePresent( "1.3.6.1.4.1.65536.0.4.3.2.1", "other", true );
        checkObjectClassPresent( "1.3.6.1.4.1.65536.0.4.3.2.2", "other", true );

        // sync operation happens anyway on shutdowns but just to make sure we can do it again
        getService().sync();

        getService().shutdown();
        getService().startup();

        checkAttributeTypePresent( "1.3.6.1.4.1.65536.0.4.3.2.1", "other", true );

        checkObjectClassPresent( "1.3.6.1.4.1.65536.0.4.3.2.2", "other", true );
    }
```

7) Defining two LdapServer

When testing replication, it's important to define two different servers. The annotations we have defined can be used for single methods which are not tests. Let's see how we design a test where two servers are created, one consumer and one provider:

```
public class ClientServerReplicationIT
{
    private static LdapServer providerServer;

    private static LdapServer consumerServer;

    private static SchemaManager schemaManager;

    @BeforeClass
    public static void setUp() throws Exception
    {
        Class<?> justLoadToSetControlProperties = Class.forName( FrameworkRunner.class.getName() );
        startProvider();
        startConsumer();
    }


    @AfterClass
    public static void tearDown()
    {
        consumerServer.stop();
        providerServer.stop();
    }

    @CreateDS(
        allowAnonAccess = true,
        name = "provider-replication",
        enableChangeLog = false,
        partitions =
        {
            @CreatePartition(
                name = "example",
                suffix = "dc=example,dc=com",
                indexes =
                {
                    @CreateIndex(attribute = "objectClass"),
                    @CreateIndex(attribute = "dc"),
                    @CreateIndex(attribute = "ou")
                },
                contextEntry=@ContextEntry( entryLdif =
                    "dn: dc=example,dc=com\n" +
                    "objectClass: domain\n" +
                    "dc: example" ) )
           })
    @CreateLdapServer(transports =
        { @CreateTransport( port=16000, protocol = "LDAP") })
    public static void startProvider() throws Exception
    {
        DirectoryService provDirService = DSAnnotationProcessor.getDirectoryService();

        providerServer = ServerAnnotationProcessor.getLdapServer( provDirService );
        providerServer.setReplicationReqHandler( new SyncReplRequestHandler() );
        providerServer.startReplicationProducer();

        // init the provider...
    }


    @CreateDS(
        allowAnonAccess = true,
        enableChangeLog = false,
        name = "consumer-replication",
        partitions =
        {
            @CreatePartition(
                name = "example",
                suffix = "dc=example,dc=com",
                indexes =
                {
                    @CreateIndex(attribute = "objectClass"),
                    @CreateIndex(attribute = "dc"),
                    @CreateIndex(attribute = "ou")
                },
                contextEntry=@ContextEntry( entryLdif =
                    "dn: dc=example,dc=com\n" +
                    "objectClass: domain\n" +
                    "dc: example" ) )
             })
    @CreateLdapServer(transports =
        { @CreateTransport( port=17000, protocol = "LDAP") })
    @CreateConsumer
        (
            remoteHost = "localhost",
            remotePort = 16000,
            replUserDn = "uid=admin,ou=system",
            replUserPassword = "secret",
            useTls = false,
            baseDn = "dc=example,dc=com",
            refreshInterval = 1000,
            replicaId = 1
        )
    public static void startConsumer() throws Exception
    {
        DirectoryService provDirService = DSAnnotationProcessor.getDirectoryService();
        consumerServer = ServerAnnotationProcessor.getLdapServer( provDirService );

        final ReplicationConsumerImpl consumer =
(ReplicationConsumerImpl)ServerAnnotationProcessor.createConsumer();

        List<ReplicationConsumer> replConsumers = new ArrayList<ReplicationConsumer>();
        replConsumers.add( consumer );

        consumerServer.setReplConsumers( replConsumers );
        consumerServer.startReplicationConsumers();

        // init the consumer...
    }

    ...
```

In this example, we have defined two servers listing on two different ports. We can also see that we have defined a specific @CreateConsumer annotations in order to configure the consumer.

III Schema aware API

One of the major issues when writing a Java application interfaced with a LDAP browser is that the existing API are not schema aware. The direct implication is clear when we get the data back from the server and we try to compare them locally.

For instance, many AttributeTypes aren't case sensible. Here is an example that expose the potential problem one may encounter:

```
    @ApplyLdifs(
        {
            // Entry # 1
            "dn: cn=Test   Lookup,ou=system",
            "objectClass: person",
            "cn: Test Lookup",
            "sn: sn test" })
    public void testLookupCn() throws Exception
    {
        LdapConnection connection = getWiredConnection( getLdapServer(), "uid=admin,ou=system",
"secret" );
        Entry entry = connection.lookup( "cn=test lookup,ou=system", "cn" );
        assertNotNull( entry );

        // Check that we don't have any operational attributes :
        // We should have only 3 attributes : objectClass, cn and sn
        assertEquals( 1, entry.size() );

        // Check that all the user attributes are present
        assertTrue( entry.contains( "cn", "Test Lookup" ) );
        assertFalse( entry.contains( "cn", "test lookup" ) );
        assertFalse( entry.contains( "2.5.4.3", "test lookup" ) );
        assertFalse( entry.contains( "CN", "  test    LOOKUP  " ) );
    }
}
```

As we can see, we are not allowed to fetch the value using another form than 'cn' (even if CommonName or 2.5.4.3 are valid identifiers for this AttributeType), and we must compare the value to what has been stored in the server, even if 'cn' is not case sensitive...

How can we proceed? The best solution is to use a Schema Aware API. Apache LDAP API provides such a mechanism. Let's see how it works...

```
    public void testLookupCnSchemaAware() throws Exception
    {
        LdapConnection connection = getWiredConnection( getLdapServer(), "uid=admin,ou=system",
"secret" );

        // Make the connection schema aware
        connection.loadSchema();

        Entry entry = connection.lookup( "cn=test lookup,ou=system", "cn" );
        assertNotNull( entry );

        // Check that we don't have any operational attributes :
        // We should have only 3 attributes : objectClass, cn and sn
        assertEquals( 1, entry.size() );

        // Check that all the user attributes are present
        assertTrue( entry.contains( "cn", "Test Lookup" ) );
        assertTrue( entry.contains( "cn", "test lookup" ) );
        assertTrue( entry.contains( "2.5.4.3", "test lookup" ) );
        assertTrue( entry.contains( "CN", "  test    LOOKUP  " ) );
    }
```

The only difference is that we have asked the connection to load the default schema. It's now possible to do some extended (and valid) comparisons like entry.contains("CN", " test   LOOKUP ").

Such an API makes it really convenient for developers to write their applications, as they don't have to take care of case sensitivity issues, as they happen when using, say, JNDI.

IV Future developments

We have some few other ideas to make it even easier to test LDAP applications and servers.

One of them is to 'reboot' the SLAMD effort, providing a mix of the SLAMD power and the Jmeter convenience. An idea would be to use Apache Directory Studio UI as a base to record a scenario, and to generate a script from it. This script will then be executed on many agents, and Studio will gather the results to provide a performance graph.

Another aspect would be to generate the associated unit tests, runnable in maven or ant.

We would also like to make the annotations able to handle another LDAP server, including OpenLDAP or OpenDJ. You will then be able to write your tests using the Apache LDAP API, but target a specific server.