Distributed Testing Framework

Users Guide

Table of Contents

Introduction

DTF is a distributed testing framework that allows for a simple yet powerful way to test distributed systems. All test cases written for execution within DTF are written in XML and have tags that will be explained in later sections of this document. To start as an introduction this document will first start by stating what is the current architecture of DTF and how it can be deployed in a couple of example scenarios.

What DTF Solves

When compared to other testing tools/frameworks **DTF** has been designed to solve a few problems that others have yet to try to solve. When comparing the **XML** language that **DTF** supports with the scripting language that most other tools support a few things become evident:

- Scripts become outdated between releases frequently and when there are changes made to the application usually this leads to have to make independent changes on each of the scripts. **DTF** solves this with its **XML** language, that can easily be transformed using a style sheet. With this transformation you can create a new set of tests or even add logic to your tests to detect if the certain tags are to execute for release 1.0 or release 1.1. Aside from the fact that this saves you a ton of man hours it also saves you from copy and paste errors, when making manual changes to hundreds of script files.

- Scripts are hard to read since most scripting languages are very extensive and allow the same thing to be done in *n* different ways. This doesn't mean that **DTF's XML** language makes it hard for you to write things, the language itself has been designed to allow you to do extremely complex things with very simple group of tags. By allowing you to nest looping tags in any which way you'd like you can easily create a complex loop with *n* different branches in less than 10 of **XML**.

- Documentation for scripting languages gets out of sync easily when there are multiple people making changes to the same **APIs**. This is due mainly to the fact that code and documentation for code rarely reside in one location. This has been easily solved in **DTF** by having the documentation for each tag present in same Java class that contains your tags code. Using **JavaDoc** comments right in the middle of the code, you are able to add examples, descriptions and format this in **HTML**. The only reason for not updating the documentation is laziness.

- Scripts aren't easily ported between environments because they allow the end user to refer to paths in the system and/or other environment specific elements that then will block you when trying to run the same tests on completely different environment. **DTF** has solved this by never allowing you to reference the file system directly except through the use of storages which abstract the location of those files within the test. The other thing **DTF** does to help changing environments is that each tag can easily be implemented to detect the environment and change its behavior slightly so that the tests suffer no changes and yet the tests execute in the new environment like they should.

- Data driven tests in a scripting language usually end up having every test writer drive their tests in their own formats and this doesn't allow data sets to be shared between tests written by different testers. **DTF** solves this by having the notion of record/query events and being able to create a cursor to an event file (this event file can be any format you'd like there is documentation on how to create your own query/record handlers). With this cursor you can traverse that data and easily drive your tests from common sources of data. Writing up a new data set will create a completely new tests scenario.

- No scripting languages ever make it easy to spawn multiple threads and be able to synchronize them at certain points without having to really know what you're doing in terms of using mutexes and **POSIX** threads and basically being a developer of some code that is prone to bugs. **DTF** has solved this problem with the existence of very simple tags that allow you to define concurrency and to be able to synchronize your events between threads not just on the same machine but over the network on a completely different component. This has also been extended so you can even exchange information between the various threads. Allowing for you to write tests in a few hours that would take you a few days to code in any other programming language just do to the nature of dealing with threads and networking related problems.

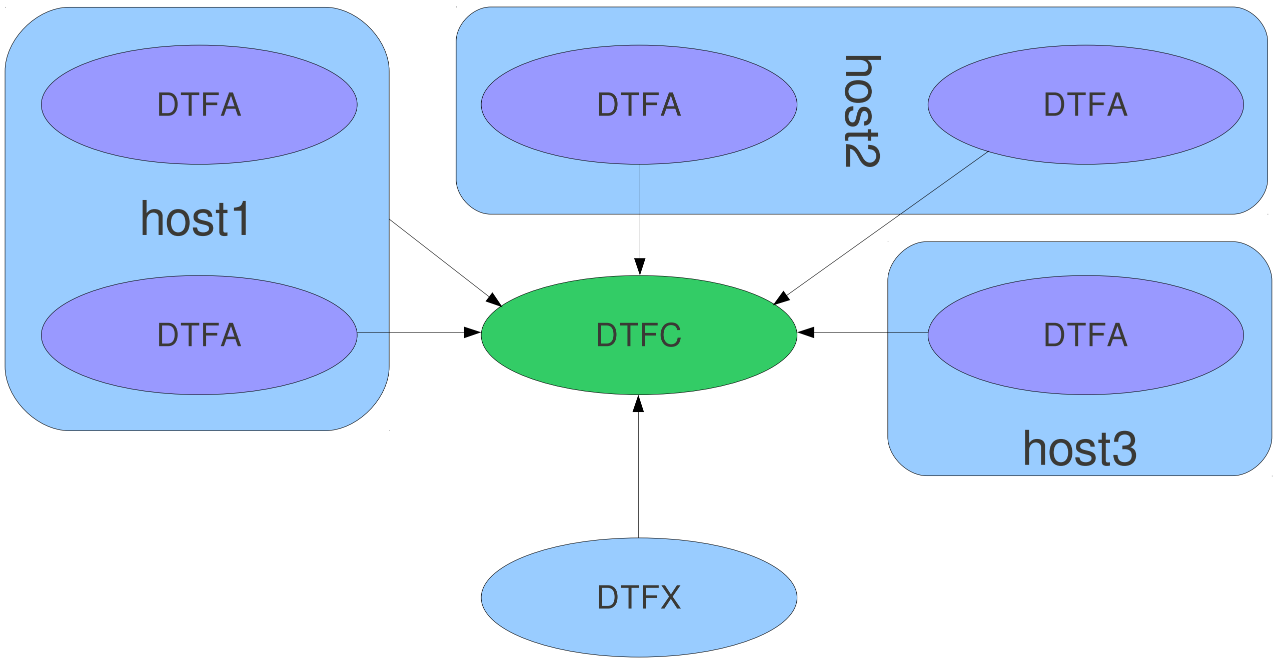
Architecture

To deploy DTF in a given environment you need to first understand it's architecture and this section will cover the architecture of DTF. There are 3 types of components in DTF the Controller (DTFC) the Agent (DTFA) and the XML Runner (DTFX). Here is small explanation of each of the components:

DTFC – Controller is the element in the framework that allows the DTFX to talk to the DTFA's and allows them to execute actions/tasks in order to fulfill the specified behavior of a test case.

DTFA – Agent is where most of the actions in a test case take place and these actions are where the framework interacts with the product being tested.

DTFX – XML Runner is the component that reads the test case written in XML and translates that into the correct execution of actions/tasks on each agent being used by the test.



The above illustration show better how the DTF architecture works by having all of the components connect to the controller (DTFC). The diagram also shows that the location of each component is really not of importance since from the test case's point of view the components are accessible through the controller. The arrows show how each of the components are responsible for connecting to the controller in order to be accessible for executing test cases.

Starting up Components

Lets take the following setup:

- 2 client machines (lets call them cl1 and cl2)

Now on each of those client machines we copy from the DTF base directory the *build/dist* directory over to each of the client machines and any other machines we may be controlling with this test. You can also use the new ant task called deploy which will push the DTF build to any Linux box that you have ssh access to.

Now on one of the client machines (cl1) will start the DTFC by executing the ***ant.sh run_dtfc*** you should see the following output:

```
> ./ant.sh run_dtfc
Buildfile: build.xml

init:
     [echo] Creating log dir logs/16-07-2008.10.11.52
    [mkdir] Created dir: /.../dtf/build/dtf/dist/logs/16-07-2008.10.11.52

run_dtfc:
     [java] INFO  16/07/2008 10:11:53 DTFNode         - Starting dtfc component.
     [java] INFO  16/07/2008 10:11:53 Comm            - Host address [cl1]
     [java] INFO  16/07/2008 10:11:53 RPCServer       - Listening at 20000
     [java] INFO  16/07/2008 10:11:53 Comm            - DTFC Setup.
```

This means the DTFC is up and running and ready to manage DTFA connections on port 20000 of the machine it was executed on (this is the default but if for some reason port 20000 is taken then it will try to bind to the next available port and then you must make sure to use the *-Ddtf.connect.port=X* on the other components to tell them to connect to the non default port). There are some properties that can be used to control the which address/port the controller will bind to. They are *dtf.listen.addr* and *dtf.listen.port* and should be specified using the normal ant property definition with *-Ddtf.listen.addr=XXX*.

So now we can start an agent and have it connect to this already running DTFC. This is simple, from the same directory as before you can execute ***ant.sh run_dtfa*** but you need to tell it where to connect to if you're running on another machine like so:

```
>./ant.sh run_dtfa -Ddtf.connect.addr=cl1
Buildfile: build.xml

init:
     [echo] Creating log dir logs/16-07-2008.09.46.10
    [mkdir] Created dir: /.../dtf/logs/16-07-2008.09.46.10

run_dtfa:
     [echo] Starting DTFA
     [echo] logging to logs/16-07-2008.09.46.10/dtfa.*.log
     [java] INFO  16/07/2008 09:46:12 DTFNode         - Starting dtfa component.
     [java] INFO  16/07/2008 09:46:13 Comm            - Host address [68.180.200.245]
     [java] INFO  16/07/2008 09:46:13 RPCServer       - Listening at 0.0.0.0:20000
     [java] INFO  16/07/2008 09:46:13 Comm            - DTFA Setup.
     [java] INFO  16/07/2008 09:46:13 CommClient      - Registering component at
http://10.73.144.127:20000/xmlrpc
     [java] INFO  16/07/2008 09:46:15 Rename          - Node name set to: dtfa-0
```

```
          [java] INFO  16/07/2008 09:46:15 CommClient      - Connected to DTFC
```

Once we see the “Connected to DTFC” line we know that this DTFA is connected to the controller and ready to tasks to execute. Now having executed one of these DTFA's from cl1 itself and another from cl2.

Now that we have the framework up and running we can use the DTFX to execute a given test case within the current DTF setup. Executing the DTFX is done like so:

```
./ant.sh run_dtfx -Ddtf.xml.filename=tests/ut/echo.xml -Ddtf.connect.addr=cl1
Buildfile: build.xml

init:
     [echo] Creating log dir logs/16-07-2008.10.23.07
    [mkdir] Created dir: /.../dist/logs/16-07-2008.10.23.07

run_dtfx:
     [java] INFO  16/07/2008 10:23:08 DTFNode         - Starting dtfx component.
     [java] INFO  16/07/2008 10:23:08 InitHook        - Example InitHook got called.
     [java] INFO  16/07/2008 10:23:08 InitPlugins     - Called init for plugin [ydht_dtf.jar].
     [java] INFO  16/07/2008 10:23:08 Comm            - Host address [cl1]
     [java] INFO  16/07/2008 10:23:08 RPCServer       - Listening at 0.0.0.0:20001
     [java] INFO  16/07/2008 10:23:08 Comm            - DTFX Setup.
     [java] INFO  16/07/2008 10:23:09 Createstorage   - Creating storage: INPUT
     [java] INFO  16/07/2008 10:23:09 Echo            - Echoing a message on the component DTFA1
```

So as simple as that we have execute the echo.xml test case from the DTF unit tests against the current DTF setup. Which only really needed one DTFA if you look at the output from the test case run we only lock on dtfa of the type dtfa and use that to do a remote echo. Here's what the remote echo looks like on the agent side:

```
          [java] INFO  16/07/2008 09:46:15 Rename          - Node name set to: dtfa-0
          [java] INFO  16/07/2008 09:46:15 CommClient      - Connected to DTFC
          [java] INFO  16/07/2008 09:46:35 CommClient      - Remote echo on component DTFA1
```

Now as you've probably noticed the components don't have to be on separate machines. This fact is used to unit test the framework from a single machine by starting all of the components DTFA, DTFC and DTFX on the same machine and running the necessary unit tests against this DTF setup. Now for other purposes such as performance testing we obviously need separate machines to be able to stress the product being tested correctly.

Deploying DTF

Deploying **DTF** to several machines and setting it up to run can be done manually and isn't that hard all you have to do is copy the ***build/dtf/dist*** directory to the machine you want to start any of the **DTF** components and then you can issue the appropriate command line to start that component up. You will need a Sun JDK 1.5+ to execute the components and then you can start them up with the following example command lines:

Start up the DTFC (Controller):

- `./ant.sh run_dtfc`

Start up a DTFA and connect it to the controller on the specified address:

- `./ant.sh run_dtfa -Ddtf.connect.addr=host_machine_for_dtfc`

Start up the test tests/xxx.xml and run it against the controller on the specified address:

- `./ant.sh run_dtfx -Ddtf.connect.addr=host_machine_for_dtfc -Ddtf.xml.filename=tests/ut/echo.xml`

That is the manual way and is easy enough for small setups with just 2-3 agents but when we get into larger setups that run multiple client side agents and a few other server side agents and we have to use **SSH** tunneling in order to connect some of those agents back to the controller machine we find that the setup time is long and prone to errors. In order to make this easier, there is a way of starting up your **DTF** setup by just passing a configuration file to a few targets in your build and requesting to check the status, stop and start your setup, etc. This takes care of all of the configuration steps for you. Lets start by showing what the format of the actual configuration file looks like:

```
<setup xmlns="http://dtf.org/v1" >
    <dtfc host="${host}" user="${user}">
        <dtfa host="${host}" user="${user}">
            <property name="node.id" value="1"/>
        </dtfa>

        <dtfa host="${host}" user="${user}">
            <property name="node.id" value="2"/>
        </dtfa>

        <dtfa host="${host}" user="${user}">
            <property name="node.id" value="3"/>
        </dtfa>

        <dtfx host="${host}" user="${user}" test="tests/ut/ut.xml"
              logs="tests/ut/output/ut_results.xml">
        </dtfx>
    </dtfc>
</setup>
```

Many of you will notice that it has a declaration just like the normal **DTF** test scripts but just a few new tags that are used to define your setup. The tags themselves just define the important properties that are necessary for the deployment feature to know where to setup each component. So you can see that you need to specify at least the **host** and **user** name to be used to **SSH** into each machine, that will also be the user that the component would be executed as. Then you can specify any properties directly in this file that would be loaded into the component at runtime.

All of the deployment targets are available under the build directory (**dtf/build/dtf/dist**) and all you need to do to see the available targets is issue the command **`./ant.sh -p`** to see each target and a description of what it does. Now we'll go through the most used targets and how they can be used to setup and monitor your **DTF** configuration.

Setup Configuration

- `./ant.sh setup-dtf -Ddeploy.config=path/to/your/config.xml`

This will setup all of the machines identified in the ***deploy.config*** file (default is *config.xml*) so that the **DTFC** and the machine issuing this command can **SSH** into any of those other machines. This allows the **DTFC** to be able to startup the necessary components on the other machines and allows the issuing machine to be able to gather logs. The **SSH** keys generated and used from now on will be under you **$HOME/.dtf** directory on *nix machines and under **%HOMEPATH%/dtf** directory for Windows. When executing this command be sure to watch for prompts like so:

```
setup-dtf:
     [java] INFO  10/11/2009 11:06:24 DeployUI        - rlgomes@dayspentlist.corp.yahoo.com:
```

This is basically a prompt from **SSH** requesting you type your password and currently the password will be visible to the person typing. **So be careful not to do this step when others are watching your screen.**

Start Configuration

```
./ant.sh deploy-start -Ddeploy.config=path/to/your/config.xml
```

The previous target starts up your **DTFC** first and then make sure that its up and running, then it will start up each of the **DTFA** specified and lastly start up a **DTFX** if one is defined. Once this has completed you should be able to check the status of your with the follow target.

Check Status of Configuration

```
./ant.sh deploy-status -Ddeploy.config=path/to/your/config.xml
```

This will output the current state of your configuration when compared with the configuration file you've specified. It will identify **DTFA's** that are no longer running as well as the state of your runner's (**DTFX**) last test run. Here's an example of running this command with the ***tests/setup/ut_config.xml*** configuration file on your localhost:

```
./ant.sh -Ddeploy.config=tests/setup/ut_config.xml -Dhost=localhost -Duser=rlgomes deploy-status
Buildfile: build.xml

init:
     [echo] Creating log dir ./logs/10-11-2009.14.19.05
    [mkdir] Created dir: /home/rlgomes/workspace/dtf/build/dtf/dist/logs/10-11-2009.14.19.05

deploy-init:

deploy-status:
     [java] INFO  10/11/2009 14:19:09 DeployDTF       - Status of dtfc on localhost
     [java] INFO  10/11/2009 14:19:09 DeployDTF       - DTFC on localhost
     [java] INFO  10/11/2009 14:19:09 DeployDTF       - DTFA [dtfa-0] on localhost is locked
     [java] INFO  10/11/2009 14:19:09 DeployDTF       - DTFA [dtfa-1] on localhost is locked
     [java] INFO  10/11/2009 14:19:09 DeployDTF       - DTFA [dtfa-2] on localhost is locked
     [java] INFO  10/11/2009 14:19:10 DeployDTF       - DTFX on localhost running
tests/ut/state_management.xml:41,16

BUILD SUCCESSFUL
Total time: 6 seconds
```

You can easily see the state of each of the components connected and that your **DTFX** is currently on the test ***tests/ut/state_management.xml*** at line 41. You can also watch the output easily with the target **deploy-watch**. The deploy-watch target will tail the output file on the **DTFX** and you can watch it on your screen and Ctrl-C that whenever you'd like. Once the status reports that the **DTFX** is completed you can easily gather your logs.

Saving your Logs

```
       ./ant.sh -Ddeploy.config=tests/setup/config.xml deploy-savelogs
```

This target simply copies back all of the component output files to the **dtf_logs** directory and will copy any other of the log files from the runner that were specified with the attribute **logs** (comma separated list). Your old **dtf_logs** will be backed up to the **dtf_logs_bk** in case you required them for some reason.

Using Components

Component tags are what differentiate DTF from something like **Ant** this because here is where the real magic is done and the writer of test cases is allowed to execute tasks remotely on other components without having to worry about how to communicate with that component and making the writing of the test case a much simpler and straightforward task. This is the section will show you how to use the various component tags described in the previous section and will attempt to give you the first steps in writing test cases with DTF.

Outputting a message on another component

Simple example on how to lock a component and then output some text to this component using DTF. this example was covered already in the Deployment section, and all can execute this test on a single machine or on 3 machines to have a physical separation of all the DTF components involved. For the purpose of this example I will run all 3 components on the same machine because it's much easier to setup and execute the test case this way.
DTF setup:

- 1 machine with a copy of the DTF build.
- Run 1 dtfc, 1 dtfa on that machine.

Now that we have those components up and running lets write our test case from scratch. So the very first thing that is necessary is the root element of the XML which is script like so:

```
<?xml version="1.0" encoding='UTF-8'?>
<script name="echo" xmlns="http://dtf.org/v1">

</script>
```

Now if you have a good XML editor with the doctype specified at the top of the XML file, the editor will aid you along with writing the test case by filling in necessary attributes and tags. Some good editors to use when writing test cases will be identified in the XML Editors section.

So looking at the previous XML we can see that we also identify the XSD being used, this XSD is used by the editor to know exactly what type of document is being handled and which elements and attributes to expect. The XSD also lets the DTFX know when an test case is well formed and when it doesn't follow the correct syntax.

The above XML is far from complete, so lets see what else is needed so there is an info section that we need to add that should describe the test case and as well identify the author of this test case for future reference. Now adding those sections this is what we have:

```
<script name="echo">
```

```
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rodney.gomes@sun.com</email>
            </author>
            <description>This test case will execute a remote echo on a DTFA. </description>
        </info>
    </script>
```

Now we need to lock a component that we can use to do the remote echo, this is done by using the tag lockcomponent to identify the type of component and the internal alias to assign to this component for future reference within the test case. Here's the current state of our test case:

```
    <script name="echo">
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rodney.gomes@sun.com</email>
            </author>
            <description>DTF echo unit test. </description>
        </info>

        <local>
            <lockcomponent id="DTFA1" type="dtfa"/>
        </local>
    </script>
```

There's only one thing really missing and that is the call to the component that will do the remote echo on it and in so, it will achieve the goal of this test case. Finished test case:

```
    <script name="echo">
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rodney.gomes@sun.com</email>
            </author>
            <description>DTF echo unit test. </description>
        </info>

        <local>
            <lockcomponent id="DTFA1" type="dtfa"/>
        </local>
        <component id="DTFA1">
            <echo>Remote echo on component DTFA1</echo>
        </component>
    </script>
```

So there we have it a simple test case that locks a DTFA and executes a remote action (echo) on this component.

Now that we do have this test lets think about a different test where we want to do the same thing as identified above but sometimes we want to do this remote action on a DTFA of the type dtfa but other times we want to run against a DTFA of client type. How do we go about this ? Well very simply put we must use properties to be able to specify them from the DTFX command line and change them at run time so the above example would now look like this:

```
    <script name="echo">
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rodney.gomes@sun.com</email>
            </author>
            <description>DTF echo unit test. </description>
        </info>

        <local>
            <lockcomponent id="DTFA1" type="${dtfa.type}"/>
        </local>
        <component id="DTFA1">
            <echo>Remote echo on component DTFA1</echo>
        </component>
    </script>
```

And to specify the type from the command line it's as simple as doing the following:

```
$./dtfx.sh cl1 9999 tests/ut/echo.xml -Ddtf.type=mydtfa
```

Now that solves the problem of being able to specify the DTFA type from the command line but it still doesn't have a default value for the DTFA. The way to have default values is to load these properties from the properties file using the loadproperties tag, and not set the overwrite attribute to true. So now the test would look like so:

```
    <script name="echo">
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rodney.gomes@sun.com</email>
            </author>
            <description>DTF echo unit test. </description>
        </info>

        <local>
            <createstorage id="INPUT" path="${dtf.path}/tests/ut/input"/>
            <loadproperties uri="storage://INPUT/ut.properties"/>

            <lockcomponent id="DTFA1" type="${dtfa.type}"/>
        </local>
        <component id="DTFA1">
            <echo>Remote echo on component DTFA1</echo>
        </component>
    </script>
```

Now I've squeezed in a little more than just the loadproperties tag because in order to refer to a file from within a **DTF** test case we need to specify a **URI**. So now for a better understanding of URI's and how they are used within **DTF** you should go to the File Access section of this document.

Using Events

Events in DTF can be thrown, recorded and queried within the test case. These events are thrown from within the code in some instances to measure the actual performance of the framework and can also be thrown using the event tag from within the test case to measure performance of certain tasks that the test case is trying to analyze. First I'll present each of the tags used within DTF to record and query events and then I'll show some complete examples and how to use the various recorder outputs to measure performance and write more complex test cases.

Simple record and query test case

The simplest test case that we can probably develop here is one similar to the ***query.xml*** unit test that exists for DTF. So lets start by creating a simple loop, and then within that loop throwing some events that we can record to a text file and later we can get those same events back with a query and show how the whole event system works together.

```
<script name="record_n_query">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rodney.gomes@sun.com</email>
        </author>
        <description>DTF record and query example. </description>
    </info>

    <local>
        <echo>Generating some events...</echo>
    </local>

    <record type="txt" uri="storage://OUTPUT/myevents.txt" append="false">
        <for property="iteration" range="[1..10]">
            <event name="dtf.myevent">
                <attribute name="iteration" value="${iteration}"/>
                <attribute name="timestamp" value="${dtf.timestamp}"/>
            </event>
        </for>
    </record>
</script>
```

So there we have a simple loop, where we generate 10 events that have two properties. One of those properties is the iteration for that event and the other one is the times tamp given to us by the ***$ {dtf.timestamp}*** property. Lets query the text file for all of the events who's iteration value are less than 6, now here is how we would do this:

```
<script name="record_n_query">
    ...

    <record type="txt" uri="storage://OUTPUT/myevents.txt" append="false">
        <for property="iteration" range="[1..10]">
            <event name="dtf.myevent">
                <attribute name="iteration" value="${iteration}"/>
                <attribute name="timestamp" value="${dtf.timestamp}"/>
            </event>
        </for>
    </record>

    <local>
        <echo>Retrieving all of the values from the events file.</echo>
    </local>

    <query type="txt"
           event="dtf.myevent"
           uri="storage://OUTPUT/myevents.txt"
           property="element">
        <select>
            <field name="timestamp"/>
        </select>
        <where>
            <lt field="iteration" value="6"/>
        </where>
    </query>
</script>
```

Now we've queried the text file but we have to iterate through the values we just got back and print them, like so:

```
    ...
        <try>
            <sequence>
               <for property="n" range="[1..6]">
                   <local>
                        <echo> Value from db: ${element.iteration}</echo>
                   </local>
                   <nextresult property="element"/>
               </for>

                <!-- should never make it here because there are only 4
                     elements but 5 iterations -->
                <local>
                    <fail message="This should never be seen."/>
                </local>
            </sequence>
            <catch exception="org.dtf.exception.NoMoreResultsException">
                <sequence>
                    <local>
                        <echo>All done iterating.</echo>
                    </local>
                </sequence>
            </catch>
          </try>
    </script>
```

So as you can see we create another loop slightly bigger than we expect, and then we use the nextresult to move the cursor of our query along, until it hits the ***NoMoreResultsException*** and then we terminate correctly. For this example the type of loop we use is from 1 to 6 since we know we only have 5 elements that are lower than 6, but for the general case we don't know how many elements we are expecting and for that we would use a different type an infinite loop and then catch the ***NoMoreResultsException*** in the same manner.

Using Cursors

What are cursors ?

Cursors are the way we can get data that was previously recorded with the ***record*** tag. Basically it allows you to iterate through the events that were previously recorded and do something with them. In most cases we'll just pass the cursor to another tag such as the ***stats*** or ***graph*** tag which will iterate over the results and process them for us. There are some cases where we'd like to iterate over our events and validate they have the right values, or we want to use those events to generate some other action during the test execution.

How to use cursors

Lets have a look at a simple test that records the ***http_post*** events and then later uses these to generate some ***http_get*** requests that would get the data back and validate its correctness. So here's the code that would actually do the ***http_post*** and record the events:

```
<record uri="storage://OUTPUT/http_put_events.txt" append="false">
    <distribute property="iteration" range="1..${clients}" iterations="${iterations}">
        <http_put uri="http://localhost/webdav/file${iteration}"
                  onFailure="continue">
            <entity value="${dtf.stream(random,128,1234)}"/>
        </http_put>
    </distribute>
</record>
```

So here we have a simple distribute that will generate upto ***${clients}*** threads and cover ***${iterations}*** of doing puts to that made up URI that uses the property ***${iteration}*** to make sure that each put is unique. Now on my machine I actually have a **Webdav** server running on that exact location so this will put files with the content generated by the **DTFStream** into there so we can retrieve with the next bit of code. So here is where we create a cursor and use it to iterate through our previously recorded events:

```
<query uri="storage://OUTPUT/http_put_events.txt"
       cursor="mycursor">
    <where>
        <eq op1="status" op2="200"/>
    </where>
</query>

<iterate cursor="mycursor">
    <http_get uri="${mycursor.uri}" onFailure="fail"/>
    <if>
        <neq op1="${mycursor.datahash}" op2="${http_get.bodyhash}"/>
        <then>
            <fail>Corrupt data!!!</fail>
        </then>
    </if>
</iterate>
```

Once again this really isn't hard to read or understand,when we do our query we're being careful to only pick the operations that succeeded (status == 200) and then we use the iterate tag to sequentially iterate through each of the events and validate the data we get back from our **http_get**.

Using Compare Tag to do Validation

The **compare** tag is extremely useful to compare two sources of events and be able to validate that the data you stored at some point is the same as the data you retrieved after some testing to the system you may be testing. Using the compare tag is pretty straightforward but it requires understanding exactly what the tag does so lets look at a simple case first where we have a service that we write records to and we use a simple record id to identify each of our saved records. The next few sections we'll cover a couple of different scenarios that use the compare tag and show you how to efficiently use this tag to do validation correctly and efficiently.

Validation with a simple HTTP file server

In this section lets cover how to validate data that has been written and read from an **HTTP** file server. This server basically stores some data in a file under a certain id we give it and then allows us to retrieve this same data at a later time with just the id we gave it at store time. Lets have a look at how to create the server service itself within **DTF** (this is just for testing purposes since we don't have a real server). Using the **http_server** tag this should be pretty easy to follow:

```
<http_server port="${port}">
     <http_listener path="/write_record" method="PUT">
         <cat uri="storage://STORE/file-${http.put.headerin.recordid}"
              append="false">${http.put.body}</cat>
     </http_listener>

     <http_listener path="/read_record" method="GET">
         <property name="data"
                   uri="storage://STORE/file-${http.get.headerin.recordid}"
                   overwrite="true"/>
         <http_response status="200">
             <entity value="${data}"/>
         </http_response>
     </http_listener>
 </http_server>
```

As you can see we create an **HTTP** server on a specified port and then we create two **HTTP** listeners that are waiting for **PUT** and **GET** requests that allow the calling **HTTP** client to store and retrieve data from this **HTTP** Server. So to use this in a test case you need to start it in parallel with the load you're going to execute. The following example is from *tests/examples/validation/validation1.xml:*

```
<parallel>
     <http_server port="${port}">
         <http_listener path="/write_record" method="PUT">
             <cat uri="storage://STORE/file-${http.put.headerin.recordid}"
                  append="false">${http.put.body}</cat>
         </http_listener>

         <http_listener path="/read_record" method="GET">
             <property name="data"
```

```
                              uri="storage://STORE/file-${http.get.headerin.recordid}"
                              overwrite="true"/>
                    <http_response status="200">
                        <entity value="${data}"/>
                    </http_response>
                </http_listener>
            </http_server>

            <try>
                 <record uri="storage://OUTPUT/http_events.txt">
                    <sleep time="2s"/>
                    <for property="i" range="1..100">
                        <http_put uri="${uri}/write_record">
                            <header name="recordid" value="${i}"/>
                            <entity>${dtf.stream(random,8,${i})}</entity>
                        </http_put>
                    </for>

                    <for property="i" range="1..100">
                        <http_get uri="${uri}/read_record">
                            <header name="recordid" value="${i}"/>
                        </http_get>
                    </for>
                </record>
                 <finally>
                    <http_server port="${port}" command="stop"/>
                 </finally>
               </try>
        </parallel>
```

The previous **XML** is not for the newbies at **DTF** and should be read carefully because a lot is going on in these 30 or so lines of **XML**. In the first part of this small example we create an **HTTP** server that is listening for **HTTP PUTS** at the path ***/write_record*** and we take the incoming data along with the header ***recordid*** and use that to create a file in the storage ***STORE*** that can later be used by the other listener to do an **HTTP GET** in a similar manner. Now that we have this service that will take writes and reads its very straightforward to write the second half of the example where we write 100 records in and read those 100 records back. This is where we want to do the validation of those event and this is exactly where the ***compare*** tag comes into action. Lets have a look at what it would look like to use the compare tag for the previously recorded events:

```
<compare>
     <query uri="storage://OUTPUT/http_events.txt"
            cursor="writes"
            event="http.put"/>

     <query uri="storage://OUTPUT/http_events.txt"
            cursor="reads"
            event="http.get"/>

     <where>
         <eq op1="${reads.headerin.recordid}"
             op2="${writes.headerin.recordid}"/>
     </where>

     <validate>
```

```
            <assert><eq op2="${writes.data}" op1="${reads.body}"/></assert>
        </validate>
    </compare>
```

The tag itself uses various other **DTF** tags to define what data is going to be compared from the two event sources as well as how and what to do when validating the data. This gives full control of the comparison to the test writer while giving him/her the ability to easily define what to do when things don't match. This allows you to log a warning or throw a failure immediately after finding some sign of corruption. The only thing the ***compare*** tag is to be efficient in how it navigates through the two queries you gave it. So above you'll see two different queries, one is for the write events that were done to a specific service while the second is for a bunch of read events. We know that the data from the read has to match the writes and we know that the ***recordid*** attribute is the only way to correlate the data from both event types. So each query relates to the two different events we're going to compare ***while*** the where tag defines how we match data from each of those sources and the ***validate*** tag is where you can compare your data or do anything you'd like once you've found two records that match the criteria defined in your where statement. In the example we're just asserting that the data from the **HTTP PUT** matches the body received as a response to the **HTTP GET** we issued.

The compare tag can also be used with a single query in which you do the validation based on the fact that you know how data was generated for each record and can easily reconstruct the data at validation time. This applies even to the case above, so the previous compare could be written like this:

```
<compare>
    <query uri="storage://OUTPUT/http_events.txt"
           cursor="reads"
           event="http.get"/>

    <validate>
        <assert>
            <eq op1="${reads.body}"
                op2="${dtf.stream(random,8,${reads.headerin.recordid})}"/>
        </assert>
    </validate>
</compare>
```

The previous usage of the ***compare*** works because we know how to reconstruct the data based on some header information that we used to create the original data itself. This can almost always be achieved as long as you make sure to shove the necessary information into your headers of your **HTTP** request because **DTF** will record these headers as part of the event attributes so you can easily reference them later. This approach has a huge difference in terms of the time it takes to validate the data because running through this single file of events and validating the data against itself can be done in the order of 20,000-30,000 validations per second while the previous method of validating the write events against the read events will most likely be in the area of 10,000 validations per second.

Currently the **compare** tag handles well sequential data and is inefficient when handling randomly ordered data that is common when you write tests that access those records in random orders. There is a already a plan to fix this soon by having an indexing mechanism as we do multiple passes through the data and being able to speed up the lookup based on your ***where*** clause.

Validation with a message queue server

In this section we'll cover the issues with testing a message queue service that can receive messages for a specific named queue from multiple writers at the same time and then return those messages to any number of readers in the same random order the message was received. In this scenario we will not use HTTP headers to hold our ids and will have to encode some data write into the message that we are sending in order to do validation in a different way. So lets start by coming up with the message queue server using the **http_server** tag (test at ***tests/examples/validate/validation2.xml***).

```
<http_server port="${port}">
    <http_listener path="/push" method="PUT">
        <property name="qname"
                  value="${http.put.headerin.qname}"
                  overwrite="true"/>
        <if>
            <share_exists id="${qname}"/>
            <else>
                <share_create id="${qname}"
                              type="queue"/>
            </else>
        </if>
        <share_set id="${qname}">
            <property name="data"
                      value="${http.put.body}"
                      overwrite="true"/>
        </share_set>
    </http_listener>

    <http_listener path="/pull" method="GET">
        <property name="qname"
                  value="${http.get.headerin.qname}"
                  overwrite="true"/>
        <if>
            <share_exists id="${qname}"/>
            <else>
                <share_create id="${qname}"
                              type="queue"/>
            </else>
        </if>

        <share_get id="${qname}" blocking="true"/>

        <http_response status="200">
            <entity value="${data}"/>
        </http_response>
    </http_listener>
</http_server>
```

The above is basically a simple message server that will allow you to queue up messages using **DTF's** share point feature and then pull the data from this queue with the **HTTP GET** request on the */pull* handler. The above is just for testing purposes and allows us to easily test out the various different approaches to validating data stored/received from this service. With this **HTTP** server listening we can now design a small test that will push a few messages onto a single queue using multiple threads, here's what it may look like. This test will also pull those messages back from the server and the order of the messages will no longer be preserved. Here is what the pushing of messages to that server may look like:

```
<parallelloop property="t" range="1..${threads}">
    <for property="i" range="1..${iterations}">
        <printf args="${i},${t},${size}"
            format="%04x%04x%04x${dtf.stream(random,${size},${i}${t})}"
            property="data" />
        <http_put uri="${uri}/push">
            <header name="qname" value="master" />
            <entity>${data}</entity>
        </http_put>
    </for>
</parallelloop>
```

The previous is very straightforward and only has a small trick done by using the ***printf*** tag to create the data message that we're sending. In this scenario we've chosen not to use the **HTTP** headers to contain any of the information that we'd use later to do validation so what we can do is place some data within the message itself and that way we can still have unique messages and be able to validate their correctness at the end. With this the rest of the test has to retrieve the same messages using a similar loop with the **http_put** replace with an **http_get** tag and recording all of those events to a file for validation phase that is going to be explained next.

At this point we need to validate the data we got back does not contain corruption or is misplaced, so what we'll do is we'll first start by comparing the **http_put** events against the **http_get** events. This approach as we'll see is going to be quite slow, here's how we'd do it in DTF XML:

```
<compare>
    <query uri="storage://OUTPUT/http_events.txt"
           cursor="reads"
           event="http.get"/>

    <query uri="storage://OUTPUT/http_events.txt"
           cursor="writes"
           event="http.put"/>

    <where>
        <eq op1="${reads.body:string:sub-string(0,8)}"
            op2="${writes.data:string:sub-string(0,8)}"/>
    </where>

    <validate>
        <assert>
            <eq op1="${reads.body}"
                op2="${writes.data}"/>
        </assert>
    </validate>
</compare>
```

The above is the usual use of the **compare** tag, and the only trick in there is the use of the String Transformer to extract the first 8 characters from the data stored or retrieved so that we can compare the part of the message that contains the iteration and thread number. With that we can then validate the data across both events. Now as I mentioned this is very slow, I mean less than 100 validations per second, the reason for that is that the events in the *http_events.txt* file are not ordered by thread and therefore the compare tag spends a lot of time cycling through the events till it finds the right one. As mentioned in the previous example there is still some work necessary to make this issue less of a pain but there will almost always be a penalty for having to iterate through many events in order to find the right one. The reason there will always be a penalty is that the recorder feature of **DTF** is designed to be as fast as it can on writes and not be optimized for reads. We may review this in the future and come up with a better solution but for now this is what we have to work with.

Now the good news is that we can validate the previous data in a completely different way and that is because the write events contain all the information you need to validate the data. Lets have a look at the following use of the **compare** tag:

```
<compare>
    <query uri="storage://OUTPUT/http_events.txt"
           cursor="reads"
           event="http.get"/>

    <validate>
        <scanf args="i,t,size,data"
               format="%04x%04x%04x%${size:convert:from-hex}s"
               input="${reads.body}"/>
        <assert>
            <eq op1="${data}"
                op2="${dtf.stream(random,${size:convert:from-hex},${i:convert:from-hex}$
{t:convert:from-hex})}"/>
        </assert>
    </validate>
</compare>
```

The above **compare** tag use knows that the data contains the iteration, thread and size of the data that was generated. So using the **scanf** tag we're able to extract those values into some properties and then test the data against the same method we used to generate it. This approach is able to validate in the order of thousands of events per second. Which actually allows you to validate data from a test run in a few minutes.

Analyzing a DTF Test

The following section will walk through a test case and will go through all of the details of every aspect of the framework and what it gives a test writer the ability to do. The walk through should be your starting point before ever attempting to write your own test case and should give you a better understanding of how things are done within DTF.

So first thing lets present a very typical test case that makes use of all of the features of DTF:

```
<?xml version="1.0" encoding="UTF-8"?>

<script name="exampletest" xmlns=”http://dtf.org/v1”>
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>DTF example test that is explained in every detail within
                     the DTF User's Guide.</description>
    </info>

    <!-- Start by definig the stores will be using during this test as well
         as loading any default properties files. -->
    <local>
        <createstorage id="INPUT" path="${dtf.xml.path}/input" />
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output" />

        <loadproperties uri="storage://INPUT/test.properties" />

        <property name="runid" value="${dtf.timestamp}"/>
    </local>

    <!-- Locking components -->
    <for property="agent" range="1..${agents}">
        <local>
            <lockcomponent id="AGENT${agent}">
                <attrib name="type" value="${dtfa.type}"/>
            </lockcomponent>
        </local>
    </for>

    <record type="txt"
            uri="storage://OUTPUT/remote_event_perf.txt"
            event="dtf.perf"
            append="false">
        <!-- Execute in parallel the follow actions on those agents we
             previously locked. -->
        <parallelloop property="agent" range="1..${agents}">
            <component id="AGENT${agent}">
                <!-- On each of those agents thread this ${treads} times
                     and then do a loop of 1..${iterations} sequentially in
                     each thread. The base action is just to throw the event
                     child that you see below identifying the runid,iteration
                     and a timestamp value for testing purposes. -->
                <parallelloop property="thread" range="1..${threads}" type="parallel">
                    <for range="1..${iterations}" property="iter">
                        <event name="dtf.perf.echo">
                            <attribute name="runid" value="${runid}"/>
                            <attribute name="iteration" value="${iter}" />
                            <attribute name="timestamp" value="${dtf.timestamp}" />
                        </event>
                    </for>
                </parallelloop>
            </component>
        </parallelloop>
    </record>

    <!-- Querying the results generated above now we can generate statistics -->
    <query uri="storage://OUTPUT/remote_event_perf.txt"
           type="txt"
           event="dtf.perf.echo"
           cursor="perfcursor" />

    <record type="object" uri="property://dbperf">
        <stats cursor="perfcursor" event="stats" />
    </record>

    <!-- Print the results from the previously analyzed results -->
    <local>
        <echo>
            Remote Event performance on ${agents} agent(s) with ${threads} thread(s).
            Events per second: ${dbperf.avg_occ}
            Total of Events:   ${dbperf.tot_occ}
            Total Duration:    ${dbperf.tot_dur}s
        </echo>
    </local>
</script>
```

Now lets start decomposing each part of the test and going into the details of why and how things are done within the DTF framework. So all tests start with an XML declaration which is common to all XML files that will allow editors to know what the XSD that is being used. This XSD is used to validate the structure of the XML file even before the framework itself has done anything with the test. The declaration is always the same and looks like so:

```
<?xml version="1.0" encoding="UTF-8"?>
<script name="exampletest" xmlns="http://dtf.org/v1">
```

After this we have the root of all tests written in DTF, this root is the **script** XML node and it only has 1 required attribute which is **id**. The id attribute is used internally to identify which test is running at any given moment, it should be unique with in the test suites so that you can easily identify any of the test results by that id. After the **script** node there is a required **info** node that will describe the test and identify the person who wrote this test. Here's the part I'm talking about:

```
<script name="exampletest">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>DTF example test that is explained in every detail within
                     the DTF User's Guide.</description>
    </info>
```

This section will be very important once you start requiring test inventory list and want to know what each of the tests does (check the **Test Inventory** section for more information).

Now the test starts to actually do more interesting activities. At the top of each test you will start by defining the storages you want to use through your test case. Normally all tests have an input storage and an output storage, they are defined like so:

```
    <!-- Start by definig the stores will be using during this test as well
         as loading any default properties files. -->
    <local>
        <createstorage id="INPUT" path="${dtf.xml.path}/input" />
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output" />
```

Storages are one of the basic elements of any test and they are the only location that you are allowed to get inputs from or generate outputs to. More information on Storages can be found at File Access section. For those paying attention you may have noticed the reference to a *${dtf.xml.path}* and the *${dtf.timestamp}* properties and wonder where this comes from, well there are a small collection of properties that exist by default that are given to the test writer by the framework. These properties are useful for getting random numbers within the framework or being able to figure out where the test is executing. More information on the internal properties can be found in the **Internal Properties** (XML docuemntation) section.

At this point you'll notice the *loadproperties* tag is being used to load default properties into this test case. This tag uses the uri attribute to refer to a file in one of the previously defined storages. The load properties tag will not overwrite by default and the reasoning for that is that for most of those properties you want the ability to have the command line take precedence, so the properties loaded by the *loadproperties* tag are usually default values that we may change for different execution environments. Here's the specific section we were just discussing:

```
        <loadproperties uri="storage://INPUT/test.properties" />

        <property name="runid" value="${dtf.timestamp}"/>
    </local>
```

I did add the property tag in there so I could go into what properties are used for. Within the XML test cases properties are the variables of XML language. They allow you to define a property(variable) at the top of your test and use it through out your test. They allow you to redefine properties during runtime to control the flow of your application (remember to use the overwrite attribute to overwrite any existing value). Property resolution has been created in a way that you can refer to properties who's values are again another property so you can actually do the following: *${prop${index}}* and that will resolve the index property first and then the value of the outer property will be resolved. This gives you the possibility to create dynamically named properties which can have some pretty advanced usages.

At this point in the test we start allocating the resources for testing that we are going to need for the rest of the test execution. The resources that we usually need to allocate are DTFA's (Distributed Testing Framework Agents). These agents will be used to execute the remote actions that we need to execute in order to perform the specific behavior of our test case. Now lets have a look at the specific part of the test I'm referring to and go into more detail on what is going on here:

```
    <!-- Locking components -->
    <for property="agent" range="1..${agents}">
        <local>
            <lockcomponent id="AGENT${agent}">
                <attrib name="type" value="${dtfa.type}"/>
            </lockcomponent>
        </local>
    </for>
```

Now the above locking is a little more complex that the type that is used in most cases. Normally you know upfront the agents you're going to be using and would just write 1 *lockcomponent* tag per agent that you're going to want. In this example I've chosen to show some of the more advanced usages so that people can start to see how flexible and powerful the language itself really is. So above you'll see that this test can lock up to *${agents}* agents when it executes. This allows you to run the same test from 1 agent or N agents depending on what you set the property to. The type of agent we're looking for has the attribute type equal to *${dtfa.type}* which is set to dtfa in the properties file that we loaded earlier. For more information on components and locking have a look at the **Locking Components** Section.

So at this point we're past the setup part of the test and are now going to start using the locked components to actually do some work. We'll start by pasting the block that does most of the work in this test case and then start decomposing that into smaller parts that we can analyze and understand.

```
    <record type="txt"
            uri="storage://OUTPUT/remote_event_perf.txt"
            event="dtf.perf"
            append="false">
        <!-- Execute in parallel the follow actions on those agents we
             previously locked. -->
        <parallelloop property="agent" range="1..${agents}">
            <component id="AGENT${agent}">
                <!-- On each of those agents thread this ${treads} times
                     and then do a loop of 1..${iterations} sequentially in
                     each thread. The base action is just to throw the event
                     child that you see below identifying the runid,iteration
                     and a timestamp value for testing purposes. -->
                <parallelloop property="thread" range="1..${threads}">
                    <for range="1..${iterations}" property="iter">
                        <event name="dtf.perf.echo">
                            <attribute name="runid" value="${runid}"/>
                            <attribute name="iteration" value="${iter}"/>
                            <attribute name="timestamp" value="${dtf.timestamp}"/>
                        </event>
                    </for>
                </parallelloop>
            </component>
        </parallelloop>
    </record>
```

So lets start from the inside out and first look at the XML node that is deepest in this stack. That would be the *event* tag, this tag does nothing but throw an event with whatever name you've given it and attach the attributes tags that are direct children of the event tag. If you happened to have an action or a sequence of actions below this tag then you would be generating the event for the whole group. Now when an event is thrown it also measures the amount of time it took to execute the underlying actions. This tag has a very similar counter part in the actual Java code that can be used by the underlying actions to measure certain parts of test execution in order to generate information about each individual action so that you can then gather these events to be able to calculate performance numbers from the previously executed actions.

Now that we just assume that the event tag is throwing some events and measuring the amount of time it took to execute such an event, we can begin to look at the two tags that surround this one inside the component tag, here's the part I'm currently talking about:

```
<component id="AGENT${agent}">
    <parallelloop property="thread" range="1..${threads}">
        <for range="1..${iterations}" property="iter">
            <event name="dtf.perf.echo">
                <attribute name="runid" value="${runid}"/>
                <attribute name="iteration" value="${iter}"/>
                <attribute name="timestamp" value="${dtf.timestamp}"/>
            </event>
        </for>
    </parallelloop>
</component>
```

So whatever is inside of the component tag is actually executed on the component that we previously locked in this test. You can read more about components in the **Using Component** Section. Within the component are the actions that we want to execute on this Agent. For this example we're executing two types of loops. One of them is a parallel which means that the underlying actual will actually execute in parallel as many times as specified in the range attribute (Read the **Range Expressions** section for more information on ranges). Inside of the parallel loop we have for of 1 to the value of the property *iterations*. So far this should have be pretty easy to read and understand that on each component we're executing *${threads}* parallel threads that sequentially execute the action *Event* exactly *${iterations}* times each one.

Now around the component tag there's a *parallelloop* tag that is actually execute the underlying tag in parallel as many times as the range attribute tells it to. This is what guarantees that all of our previously locked components will execute the same actions in parallel at the same time.

There is yet one last tag around this part of the test and that is the *record* tag that is being used to record the events being thrown from any underlying actions. This mechanism has been put in place like this so that if you're not interested in underlying events then you don't have the *record* tag anywhere, and the event throwing mechanism has the tiniest overhead on any of the underlying work being done. On our example though we are recording the results to a file in our OUTPUT storage for posterior analysis (More information on recorders can be found at the **Using Events** section).

At this point our test case has done all of the actual work and now has some event files that were recorded previously to analyze. There are a few things that can be done with these files and the more common ones are to generate the performance statistics for these files, to under stand better how things behaved from a visual perspective. The next part of the test could have been made into a separate test and could have been executed at the end on its own various times and tweaking certain values in the analysis. The only reason it was placed in here was so that we could explain the various details in 1 single test.

So first lets look at how we calculate some statistics from the previous run, here's the part of the test I am looking at right now:

```
    <query uri="storage://OUTPUT/remote_event_perf.txt"
           type="txt"
           event="dtf.perf.echo"
           cursor="perfcursor" />

    <record type="object" uri="property://dbperf">
        <stats cursor="perfcursor" event="stats" />
    </record>

    <!-- Print the results from the previously analyzed results -->
    <local>
        <echo>
            Remote Event performance on ${agents} agent(s) with ${threads}
thread(s).
            Events per second: ${dbperf.avg_occ}
            Total of Events:   ${dbperf.tot_occ}
            Total Duration:    ${dbperf.tot_dur}s
        </echo>
    </local>
```

The first thing to notice is that there is a tag that is the counterpart of the *record* tag and that is the *query* tag. This tag allows us to “query” the events that were previously recorded for either validation purposes or to generate other results of interest. In our example we start by using this query to just find all of the *dtf.perf.echo* events from the previously recorded file and use those values to generate statistics using the *stats* tag. One thing to notice is that within the framework everything that has to do with querying the event files and manipulating that data uses the notion of a cursor. This cursor is named and can be referenced by a few different tags, as well as it can be reset using the *resetcursor* tag or moved forward using the *nextresult* tag. The tags themselves all have documentation that should be sufficient to understand them and use them (generating this documentation is discussed in the **Tag Documentation** section). Now once the *stats* tag does its magic we have some properties that will contain very important values for us such as the average occurrences per second of an event (avg_occ) or the total time spent doing the specific action (tot_dur). These values are calculated by the stats tag by iterating through the cursors results and doing the appropriate math on these values.

Generating load with DTF

There are many ways of generating different load patterns with DTF and because of this I'd like to show the easiest way to do some tasks and then allow the reader of this document to make a better assessment on how he/she would like to go about using all of the different tags the framework has for generating different behavior.

Single Agent Example

Lets start by looking how to take a simple test that just echo's a single message on a component to echoing the same message on N components, so here's the base test:

```
<script xmlns="http://dtf.org/v1" name="example">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>DTF example test.</description>
    </info>

    <local>
        <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

        <loadproperties uri="storage://INPUT/test.properties"/>

        <lockcomponent id="CLIENT1">
            <attrib name="type" value="${dtfa.type}"/>
        </lockcomponent>
    </local>

    <component id="CLIENT1">
        <echo>Test</echo>
    </component>
</script>
```

Thats about the simplest test you can have in DTF that makes use of all of the components in the framework. It locks a component then uses that component to echo the message “Test” and terminates releasing any previously locked components.

Multiple Agent Example

Now how do we make the same test output to various agents instead ? Its pretty straightforward, first you need to make sure to lock all of those components and then loop over those same components and run the same action on each of them so lets look at the first solution below:

```
<script xmlns="http://dtf.org/v1" name="example">
    ...

    <local>
```

```
            <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
            <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

            <loadproperties uri="storage://INPUT/test.properties"/>
        </local>

        <for property="client" range="1..${clients}">
              <local>
                  <lockcomponent id="CLIENT${client}">
                      <attrib name="type" value="${dtfa.type}"/>
                  </lockcomponent>
              </local>
        </for>

        <for property="client" range="1..${clients}">
              <component id="CLIENT${client}">
                  <echo>Test</echo>
              </component>
        </for>
    </script>
```

So that was pretty easy, we made use of the *for* tag to loop over a range of values from 1 to *${clients}* which you can then change on the command line. We loop over this range and lock each component with its own ID and then we're able to loop in the same manner and do the action on each of the components. You may have noticed that each action on each component is executed in sequence since we're in a *for* loop. If you wanted to do those actions in parallel then we'd have to use the *parallelloop* like so (presenting just the part of the test where we're executing the echo's on the agents):

```
        <parallelloop property="client" range="1..${clients}">
              <component id="CLIENT${client}">
                  <echo>Test</echo>
              </component>
        </parallelloop>
```

This couldn't possibly get any easier, by changing the tag that surrounds the *component* tag we can change the way the underlying tag is executed. So now if we wanted to execute same action in parallel on all available components but instead of just once we wanted to do it lets say 100 times in a row, one might come up with a solution like so:

```
        <parallelloop property="client" range="1..${clients}">
            <for property="iter" range="1..${iterations}">
                 <component id="CLIENT${client}">
                       <echo>Test</echo>
                  </component>
            </for>
        </parallelloop>
```

Now you might be thinking well whats the difference between that way of doing it and the following:

```
        <parallelloop property="client" range="1..${clients}">
              <component id="CLIENT${client}">
                   <for property="iter" range="1..${iterations}">
                         <echo>Test</echo>
                   </for>
              </component>
        </parallelloop>
```

So both of these will execute the same *echo* 100 times in a row in parallel across all the components. The only difference is that the first example will need to communicate to send the echo tag to the each of the components 100 times while the second sends the for tag that includes the action to the component and this one executes that and returns. So in terms of efficiency and being able to use the components of the system correctly the second option is the appropriate one. This becomes more evidente when you start using parallelizing the tags on the component side, because at this point you'll have some 20-30 threads on each component doing work and if instead you were running those same threads on the runner (dtfx) side then your test couldn't scale because as you tried bumping up the value of *${clients}* your runner machine would have too many threads running on it.

Validating Load Results

At this point we've covered the basics on doing sequential and parallel loops and how to use these within each other, now lets move into the realm of wanting to do certain load patterns and guaranteeing the amount of load at any instant in time is exactly what we're expecting. This type of load could be ramp up load where we want the system to start off slowly and then push more and more operations at a time, or could be the case of load that has variations with time to simulate what a real world scenario would do. Lets start with the example of wanting to use N clients and doing exactly 5 operations per second on each client. For this purpose we'll introduce a new tag called *distribute* which was designed with the ability to guarantee distributions during your test execution. So lets write the test that will guarantee exactly 5 operations per second across multiple clients (lets assume the components are locked and not show that code anymore):

```
<parallelloop property="client" range="1..${clients}">
      <component id="CLIENT${client}">
          <distribute workers="2" func="const(5)" iterations="20">
                <echo>Test</echo>
          </distribute>
      </component>
</parallelloop>
```

The new tag ***distribute*** has a few attributes that I should explain at this point so the workers attribute defines basically how many threads can be used to guarantee the work we want to simulate. The number of workers depends on exactly how many threads you're willing to use, plus taking into account the amount of time each will take. The ***func*** attribute is exactly where you're able to specify the type of load you'd like to generate in this case you're saying guarantee a constant of 5 operations per unit of time (unit of time defaults to seconds, can be changed with the attribute ***units***). I decided to use the ***iterations*** attribute to limit the run of the distribute tag to exactly 20 completed executions of the underlying action, but there is also a ***timer*** attribute that allows you to base the execution on time.

At this point you maybe thinking so how do I know for certain that the expected distribution of work was achieved ? Well there are two ways: 1) You can record some events that the distribute tag throws during execution that will let you know for each instant of time what the work goal was and what was the amount of work actually done. 2) You can use an external tool such as gnuplot or your favorite spreadsheet application to generate graphs from the DTF data, look at the **Graphing your data** section for more information

So first lets start by validating the distribute tag did the amount of work we expected using the events it throws. Now be aware that the distribute tag itself throw some events only when you use the attribute ***id***, so it's important to set that attribute. In this scenario also be aware that the ***echo*** tag itself does not throw any events, unlike most tags such as ***http_get, http_set,*** etc. Another thing to also note is that tags that are built into DTF that have a Java only implementation will throw events and therefore you need to use the record tag to collect those events, but for products that use an API that is different from Java currently we have a secondary process that does the work and logs the results (in the same event format that DTF recognizes) to a secondary file and there's usually a tag that allows you to gather those results to be analyzed by the framework. So here's an example of how to record those events:

```
             <record uri="storage://OUTPUT/perf.txt" append="false">
                 <parallelloop property="client" range="1..${clients}">
                       <component id="CLIENT${client}">
                           <distribute id="dist${client}"
                                       workers="2"
                                       func="const(5)"
                                       iterations="20">
                                 <echo>Test</echo>
                           </distribute>
                       </component>
                 </parallelloop>
            </record>
```

So recording the events is straightforward, with the only thing to remember is that if you don't set the ***id*** attribute in the ***distribute*** tag then you don't get the events thrown back. The ***id*** attribute is useful because then you can make sense of which events are from which components and be able to identify the exact component that didn't meet its goal (***distribute*** tag will throw events with the name of the ***id*** attribute). After the events are recorded we need to use the ***query*** tag to create a cursor that can iterate through each of the recorded values, in this example we'll also make sure to separate each of the recordings by client so we can easily identify which did not hit the target. So lets create the cursor like so:

```
          <for property="client" range="1..${clients}">
                  <query uri="storage://OUTPUT/perf.txt"
                         cursor="c1"
                         event="dist${client}"/>
          </for>
```

Notice that I've used the distribute ***id*** to distinguish the event name in order to only process each of the events per client. Now at this point we can use the ***iterate*** tag to run through the values of the cursor and compare the fields ***workDone*** and ***workGoal*** and make sure they match up as we'd expect them to. So here's what the finished solution would look like:

```
          <for property="client" range="1..${clients}">
              <query uri="storage://OUTPUT/perf.txt"
                     cursor="c1"
                     event="dist${client}"/>
```

```
            <log>Validating the workDone vs workGoal requested events</log>
            <iterate cursor="c1">
                <if>
                    <neq op1="${c1.workdone}" op2="${c1.workgoal}"/>
                    <then>
                        <fail message="workGoal != workDone for CLIENT${client}, $
    {c1.workdone} != ${c1.workgoal}"/>
                    </then>
                </if>
            </iterate>
        </for>
```

So as you can see the work completed is equal to the work requested over the course of this test. Now if we wanted to run on multiple clients and wanted to validate multiple clients. Here are the list of things that need to happen before we can graph it in that manner:

1. The events generated by the dist should have the same name so the id attribute has to be equal to something common lets say just “dist”.
2. The events being thrown by dist should have an extra field to distinguish which client is throwing that event. That is achieved using the ***attribute*** tag just before calling the distribute to add an additional default field to all subsequently generated events.
3. The ***query*** tags need to include in the select the new client attributes

Here's what the final resulting XML code would look like:

```
    <record uri="storage://OUTPUT/perf.txt" append="false">
          <parallelloop property="client" range="1..${clients}">
                <component id="CLIENT${client}">
                    <attribute name="client" value="CLIENT${client}"/>
                    <distribute id="dist"
                                workers="2"
                                func="const(5)"
                                iterations="20">
                          <echo>Test</echo>
                    </distribute>
                </component>
          </parallelloop>
    </record>

    <query uri="storage://OUTPUT/perf.txt" cursor="workdone">
        <select>
            <field name="client"/>
            <field name="workdone"/>
        </select>
    </query>

    <query uri="storage://OUTPUT/perf.txt" cursor="workgoal">
        <select>
            <field name="client"/>
            <field name="workgoal"/>
        </select>
    </query>
```

Then you'd have to loop through each of those queries and validate the workgoal and workdone data, using simply the ***iterate*** tag.

Advanced Load Generation

Now we're ready to go dig into more complex scenarios where the load being generated isn't always the same action. There are a few ways of doing this we'll start with the most obvious way and that would be to have 2 parallel threads within a component doing different tasks, lets have a look at the following example using the ***http_get*** and ***http_post*** tags:

```
<record uri="storage://OUTPUT/perf.txt" append="false">
    <parallelloop property="client" range="1..${clients}">
        <component id="CLIENT${client}">
            <attribute name="client" value="CLIENT${client}" />
            <parallel>
                <distribute id="dist"
                            workers="2"
                            func="const(5)"
                            iterations="20">
                    <http_get uri="http://www.yahoo.com" />
                </distribute>
                <distribute id="dist"
                            workers="2"
                            func="const(5)"
                            iterations="20">
                    <http_post uri="http://www.yahoo.com" />
                </distribute>
            </parallel>
        </component>
    </parallelloop>
</record>
```

The above example is the simplest way of achieving the dual behaviour from the same component but its certainly not the best way, some of the problems with the previous solution include:

1. Doubled the number of threads, without any necessity to do so
2. Inability to control the exact pattern being generated by the sum of both actions (since they're now being controlled by separate ***distribute*** tags)
3. Repetition of code (2 distribute tags, this would only get worse as you needed to do other actions in parallel with these).

So what other options do we have ? Well there are a few one of the simplest is using the switch tag to make use of only 1 distribute but then flip between the various underlying actions based on some preset conditions, lets look at the following more elegant solution:

```
<record uri="storage://OUTPUT/perf.txt" append="false">
    <parallelloop property="client" range="1..${clients}">
        <component id="CLIENT${client}">
            <attribute name="client" value="CLIENT${client}" />
            <parallel>
                <distribute id="dist"
                            workers="2"
                            func="const(5)"
                            iterations="20">
                    <switch>
```

```
                        <case>
                            <eq op1="${dist.worker}" op2="0"/>
                            <http_get uri="http://www.yahoo.com"/>
                        </case>
                        <case>
                            <eq op1="${dist.worker}" op2="1"/>
                            <http_post uri="http://www.yahoo.com"/>
                        </case>
                    </switch>
                </distribute>
            </parallel>
        </component>
    </parallelloop>
</record>
```

Here we can see how the switch tag is very useful to do actions based on some condition being met. In this case I used the distribute worker property to do an action based on the worker number. You could even do this for worker numbers superior to action numbers by using the ***lt*** and ***gt*** conditions to do some actions based on a range of workers.

What if I want to do a determined action only a certain percentage of the time ? Well for something like this there is the choices tag which allows you to define the percentage of time spent doing a whole bunch of different actions, here's the previous example translated into a 30% ***http_get*** and 70% ***http_post*** scenario:

```
<record uri="storage://OUTPUT/perf.txt" append="false">
    <parallelloop property="client" range="1..${clients}">
        <component id="CLIENT${client}">
            <attribute name="client" value="CLIENT${client}" />
            <parallel>
                <distribute id="dist"
                            workers="2"
                            func="const(5)"
                            iterations="20">
                    <choices>
                        <choose howoften="30%">
                            <http_get uri="http://www.yahoo.com"/>
                        </choose>
                        <choose howoften="70%">
                            <http_post uri="http://www.yahoo.com"/>
                        </choose>
                    </choices>
                </distribute>
            </parallel>
        </component>
    </parallelloop>
</record>
```

There you can see a very easy to read example of how to generate parallel load of only 5 operations per second from N components doing 30% HTTP Get requests and 70% HTTP Post requests.

Capacity Planning

This section will be dedicated to helping figuring out how to do capacity planning for your client machines when using **DTF**. The idea is to be able to figure out what is the maximum amount of parallel executions (ie threads) that can be done on a single box executing a certain operation or group of operations and know for certain that you're not bottle-necked on the client side and are getting the most out of your client machine while stressing the server machine. The following approach may not be a *“one size fits all”* solution but it should be close enough to what someone would need to to measure the maximum load that a given client machine can generate.

Lets start by creating a test that can execute a certain action (for our testing purposes we'll just generate an empty event named *myop*) in a **distribute** with a configurable number of threads and be able to loop on that to make sure that the **distribute** slowly increases the number of threads from 50 to 100 every 1000 iterations. Here's the first simple approach to writing this test:

```
<script xmlns="http://dtf.org/v1" name="capacity">
    ...

    <for property="threads" range="50,75,100">
        <distribute id="mydist"
                    iterations="1000"
                    range="1..${threads}"
                    property="iteration">
            <event name="myop"/>
        </distribute>
    </for>
</script>
```

A very simple **for** loop that defines the thread count used by the **distribute** executes the iterations specified with the number of threads given to it by the property `${threads}`. Notice the **distribute** has no ***func*** attribute because it does not want to throttle the work being done and instead will use the available threads to complete as many executions as possible.

For this fake testing scenario we'll start with 10 threads and increase by 10 each iteration that we find that the previous performance was still less than the current. Instead of an empty **Event** tag we will fill this with a **sleep** tag that can vary between 1-5ms allowing for a more real world like scenario of executing an operation. Now the test starts to gain life:

```
    <property name="threads" value="10"/>
    <property name="prev.avg_occ" value="0"/>

    <while>
        <true/>

        <record type="stats">
               <distribute id="mydist"
                           iterations="10000"
                           range="1..${threads}"
                           property="iteration">
                   <event name="myop">
```

```
                    <sleep time="${dtf.randomInt(1,5)}"/>
                </event>
               </distribute>
        </record>

        <if>
            <lt op1="${myop.avg_occ}" op2="${prev.avg_occ}"/>
            <then>
                <log>
                All done at thread count ${threads}
                with performance of ${myop.avg_occ}
                </log>
                <break/>
            </then>
            <else>
                <log>
                Still increasing performance with ${threads} thread(s)
                with performance of ${myop.avg_occ}
                </log>
                <property name="prev.avg_occ"
                          value="${myop.avg_occ}"
                          overwrite="true"/>
                <add op1="${threads}" op2="10" result="threads"/>
            </else>
        </if>
    </while>
```

The first thing you'll notice is that I'm using a **while** loop to keep looping while true. This loop will only stop if we hit the condition that our current average operations per second (`${myop.avg_occ}`) is lower than the previous average operations per second (`${prev.avg_occ}`). At that point we'll use the **break** tag to break out of the **while** loop and the test will be completed, having found an estimate for the amount of threads that max out performance for that specific action.

From my test runs I was getting that somewhere around 120 threads the system couldn't get anymore performance out of the specific action being executed here. For more stable numbers you should really run the test for a certain amount of time and not iterations, when using iterations as your stopping point this can sometimes lead to less stable numbers and will also increase the total runtime necessary. If you use the **timer** attribute on the **distribute** tag instead and pick an interval of time that you're comfortable with measuring the performance of the specific action you're executing then you'll be able to get a stable number for each execution.

You will find a few capacity planning tests in the **dtf/tests/examples/capacity** directory of which the above mentioned example is the *capacity1.xml* which is a very easy to read test that you can follow along with the previous write up. There is another capacity test in there called *capacity.xml* and that one is a more generic test that has been given quite a few properties that can be used to control the test scenario that better suits your needs. Have a look at the **description** section of that test for more information on how to set the various test properties.

Graphing your data

There are no tags that will automatically generate graphs for you in DTF but instead there are tags that can be used to take your event data and put it into a format that is a few steps away from being graphed the way you'd like it. The reason for the framework not generating the graphs directly is that this allows you to overlay the data that you want together in any format or other application that you'd like to use for presenting test results. This ends up being more useful since DTF is not trying to replace any existing reporting tools and instead being able to output results in a format that can easily be plugged into a reporting tool that is already in use.

So as an example this is how you would use the stats recorder in order to have your statistics available immediately after running a test and also to have the CSV output for the histogram of the durations of your data:

```
<record type="stats">
    <for property="index" range="1..10000">
              <event name="event">
                  <attribute name="constant" value="1"/>
                  <attribute name="int1" value="1234567890"/>
              </event>
    </for>
</record>

<log>
Stats Recorder Numbers:
*************************
Events/second: ${event.avg_occ}
Min Duration:  ${event.min_dur}
Max Duration:  ${event.max_dur}
Avg Duration:  ${event.avg_dur}
</log>

<cat uri="storage://OUTPUT/plot.data" append="false">${event.csv_dur}</cat>
```

This small example is generating its own events with just two fields in it, but in your case you'd be running your actions underneath that first ***record*** tag. You'll also notice you have to output the ***${event.csv_dur}*** property to an external file that later will contain the following example data:

```
0,9919
1,74
2,1
4,1
10,1
14,1
21,1
41,1
76,1
```

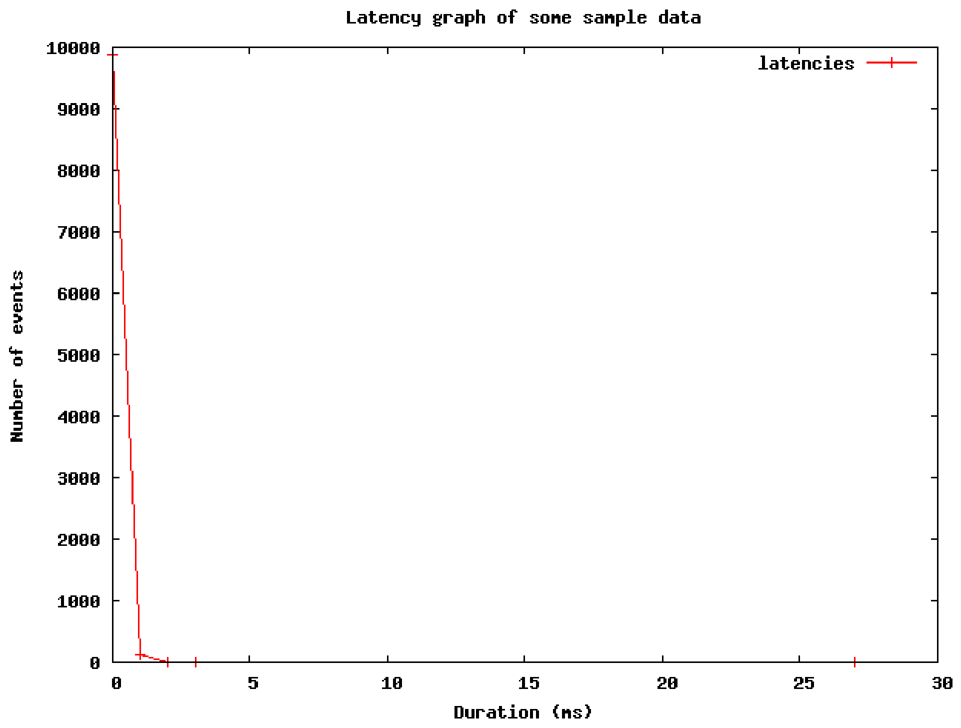
And there you have your CSV file that easily be imported into another application and used to generate whatever type of visual representation you feel adequate to your situation. This feature is very simple for the time being but there is some work being done to make this same recorder capable of sampling/interpolating the data down to just a small fraction of its original size. This new data would allow you to easily generate graphs of when the events happened without having to take into account all of the events you generated, basically allowing you to take a week long run that generated a few hundred million events and graph it as a graph only containing some 10-100 thousand events.

Using gnuplot to graph our data

Taking the previous data it is quite to use gnuplot to create a graph from it. So here's what out gnuplot script would look like:

```
set title "Latency graph of some sample data"

set terminal png
set output "graph.png"

set xlabel 'Number of events'
set ylabel 'Duration (ms)'

set datafile separator ','

plot 'tests/ut/output/plot.data' using 1:2 with linespoints title 'latencies'
```

With this gnuplot script and the output of your plot.data in the right location just running "gnuplot < gnuplot" will generate the following:



So you can see how easily it was to generate a graph using gnuplot to process our existing data so that we can visually understand that the majority of our events happened under 1ms while some few hundred were outside of the 1ms realm.

Using Storages

Storages are an important part of your tests and they are the only way you can read files from a certain location or write to files in a certain location. Currently **DTF** has only 1 type of storage available and those are storages that point to a path on the file system of the machine where you create the storage. So to create a storage you can just the existing tag **createstorage** like so:

```
<createstorage id="OUTPUT" path="${dtf.xml.path}/output" />
```

As you can see you create a storage that will be known by the id **OUTPUT** and that storage will point to the location identified by the attribute path. Now one thing to note here is that this same storage can be used for input or output as the following tags demonstrate:

```
<cat uri="storage://OUTPUT/file1">Hello World</cat>
<loadproperties uri="storage://INPUT/ut.properties"/>
```

You'll start to notice that when you reference storages with the **uri** attribute you have to use the storage:// prefix to identify the protocol so that DTF correctly understands how to get to this identity. The reason for this is because in the future if we want to allow test writers to easily pull files from an ftp or http location then they can identify that location right there in the **uri** attribute.

Now when you need to use storages on a another agent there is one small thing to do in the test when you create this storage. This is to use the attribute **export** and set it to true, with this DTF will automatically make sure that each component has its own copy of that specific storage and can easily read/write to that same storage. The only thing to note is that if you write to a common file from multiple agents in parallel the result is the same as the result you would get if you made a parallel on the runner side and wrote to it. So in the case of parallel with all of your writes being appends you'd have the various entries into that file in the same random order than the threads are running at, while if you did it sequentially the file would contain the various elements in that same order. At runtime if two components are writing and reading from the same file those files do not magically propagate their changes between the components, this can be changed in the future but for now we went with the easiest and more efficient method. Looking at the following example:

```
<log>Parallel Appends</log>
 <for property="i" range="1..10">
     <cat uri="storage://OUTPUT/remotefile" append="false"/>

     <parallel>
             <component id="DTFA1">
                 <cat uri="storage://OUTPUT/remotefile">[1:${i}]</cat>
             </component>

             <component id="DTFA2">
                 <cat uri="storage://OUTPUT/remotefile">[2:${i}]</cat>
             </component>

             <component id="DTFA3">
                 <cat uri="storage://OUTPUT/remotefile">[3:${i}]</cat>
             </component>
     </parallel>

     <property name="test" uri="storage://OUTPUT/remotefile" overwrite="true"/>
     <switch>
```

```
            <case>
                <match source="${test}"
                       expression="((\[1:${i}\])|(\[2:${i}\])|(\[3:${i}\]))*"/>
            </case>
            <default>
                <fail>${test} does not match regular expression.</fail>
            </default>
        </switch>
    </for>
```

The file *remotefile* will contain the entries **[1:${i}],[2:${i}]** and **[3:${i}]** in a different order every time the **parallel** block is executed but we know for a fact that each of those **cat** calls are done with **append="true"** so we know that all 3 entries will appear in the file on each iteration and thats why we use the regular expression to match those 3 entries in any particular order.

The same script but without the **parallel** tag we'd know the file would contain those outputs in the same order every time and could easily check like so:

```
    <for property="i" range="1..10">
        <cat uri="storage://OUTPUT/remotefile" append="false"/>

           <component id="DTFA1">
               <cat uri="storage://OUTPUT/remotefile">[1:${i}]</cat>
           </component>

           <component id="DTFA2">
               <cat uri="storage://OUTPUT/remotefile">[2:${i}]</cat>
           </component>

           <component id="DTFA3">
               <cat uri="storage://OUTPUT/remotefile">[3:${i}]</cat>
           </component>

           <property name="test" uri="storage://OUTPUT/remotefile" overwrite="true"/>
           <switch>
               <case>
                   <match source="${test}"
                       expression="\[1:${i}\]\[2:${i}\]\[3:${i}\]"/>
               </case>
               <default>
                   <fail>${test} does not match regular expression.</fail>
               </default>
           </switch>
    </for>
```

As you can see the use of remote storages has no difference when it comes to the way you use the tags in the **XML**, but you do have to remember to export the storage otherwise **DTF** will throw an exception at runtime stating that you've tried to access an inexistent storage. This feature is mainly used for being able to read common files and use those in tags you've developed which require input from files you've previously created to drive your tests behavior.

Stream Generators

Stream generators give us the ability to generate data on the fly during test execution, they take advantage of the ability to use pseudo random data or repeated data to be able to always reproduce the same data for the same signature. Currently we only support random and repeated data but there will be generators for XML, JSON, etc. The usage is very straightforward you can reference a DTFStream using the following property: `${dtf.stream(stream_type,size,argument)}`. This property will generate the data when referenced internally in the implementation of any tag. For those tags that want to really stream data and not have to resolve the property down the exact bytes described in the generator you can use internally *replacePropertiesAsStream* instead of *replacePropreties* function and you'll have access to an **InputStream** that can be used at runtime to do the right streaming activity. For an example look at the HTTP tags and you'll find that they already use the Streaming feature to make it possible to stream gigabytes of data without ever containing this data in memory.

Currently we support 2 types of stream generators:

- random
  - example: ${dtf.stream(random,32,1234)} would generate a 32 byte long stream of data with the random seed of 1234. This seed makes it possible to save the signature and validate the data later on on retrieval of this same data without ever having save the many bytes of data sent.

- repeat
  - example: ${dtf.stream(repeat,1024,BABA)} would generate 1KB of data that would have repeated BABA strings within it. Again the signature used here will always generate the same output which makes these generators really useful.

Test Inventory

DTF has some mechanisms for generating test inventory listings from the existing tests and using a very simple **XSL** stylesheet located at *dtf/src/xsl/inventory.xsl*. This style sheet can be executed by just calling the *ant inventory* target on the **DTF** framework after having built it with your specific plug-in. At this point the test inventory will be in a single file located at ***dtf/build/dtf/dist/tests/inventory.html***. This feature is just an example of what can be done and certainly the inventory list itself can be made to look a lot nicer including direct links to the **XML** or results of having run this test in the past.

File Access

All file access in DTF is done by referring a URI (Universal Resource Identifier), this is done like this because it allows for many types of protocols to be supported within the URI format. For example currently there exists on the local storages which can be created using a *createstorage* tag and can be referred to in a URI using the following syntax `storage://STORAGE_ALIAS/filename`, now if we have common files that we would like to access from various test cases but we want to access these files over http, ftp or any other protocol, well at a later date we can add support for these protocols and easily refer to these protocols within the URI format.

Communication between Components

Why communicate between components ?

There are a few reasons why one would want to communicate between components in DTF but the main one is to coordinate work in a better way to simulate a more customer like behavior or trigger certain situations on the server side that are sometimes extremely hard to hit. Being able to have a group of agents writing to the system while another group of agents awaits information on what can be read back is also a very common scenario which would allow you to notify your readers that they can now start reading back data up to a certain row count, or record number, etc. All examples covered in the following sections are in the **tests/examples/communication** folder.

How to communicate between components ?

There are currently a few ways to communicate between components the first one involves the ability to synchronize threads across components. This synchronization is achieved using the rendezvous points available in DTF. Here is how rendezvous points are used to synchronize a few threads running in parallel to synchronize every 100 executions and end up starting their work at the same time. The following example can be found at **tests/examples/communication/rendezvous1.xml**.

```
    <rendezvous_create id="rv" parties="${parties}"/>
    <parallelloop range="1..${parties}" property="thread">
        <sequence>
            <for property="iter1" range="1..5">
                   <for property="iter2" range="1..100">
                       <!--  do something -->
                   </for>
                   <rendezvous_visit id="rv"/>
            </for>
        </sequence>
    </parallelloop>
```

Your rendezvous point needs a unique name to be identified and also you need to specify how many parties will be using this rendezvous point. The parties defines the number of threads that need to visit the rendezvous before it will release them all at the same time. The visit tag will block until all parties have reached or if the the ***timeout*** attribute has been defined it will also timeout with **RendezvousException** stating that the timeout was reached. In this example we were merely synchronizing threads on the same machine but if instead of our ***sequence*** tag we had a ***component*** tag and were executing those same loops and ***rendezvous_visit*** tags on an agent the code would behave the same except that the synchronization between the threads could be off by the average network latency between these agents. If these agents have no direct connection then the communication goes through the controller, in the field testing I've done this latency has been no more than 100ms in the worst of cases.

Now the other form of communication between components in DTF involves the ability to pass actions to execute on another machine. So any component can easily pass a group of properties to set on another component, or pass a range to be recreated on another agent with a different group of values. The tags themselves are very simple but its what they're doing that can sometimes become a bit hard to understand so lets start with a simple example where we'll create scenario were we'd like to have a producer thread writing to a **Webdav** service and making sure to let another thread running that the latest file that was uploaded to the **Webdav** folder was X and that you can now retrieve that object. The second reader thread will never really stop doing work but will keep just randomly picking files that the writer tells it are available and retrieving them to make sure they're accessible. In this example we'll also have the writer notify the reader that his work has completed using the rendezvous point to send this signal.

```
<rendezvous_create id="ALLDONE" parties="2" />
<share_create id="MYSHARE" />
<parallel>
    <sequence>
        <for property="i" range="1..1000">
            <http_put uri="http://localhost/webdav/file${i}">
                <entity value="${dtf.stream(random,32,12345)}" />
            </http_put>
            <log>wrote file${i}</log>
            <share_set id="MYSHARE">
                <createrange name="files" value="random(1..${i})"
                    recycle="true" />
            </share_set>
        </for>
        <rendezvous_visit id="ALLDONE" />
    </sequence>
    <sequence>
        <try>
            <while>
                <not>
                    <rendezvous_check id="ALLDONE" />
                </not>
                <share_get id="MYSHARE" blocking="true" />
                <for property="i" range="1..10">
                    <property name="file" value="file${files}"
                        overwrite="true" />
                    <http_get uri="http://localhost/webdav/${file}" />
                    <log>read ${file}</log>
                </for>
            </while>
            <finally>
                <rendezvous_visit id="ALLDONE" />
            </finally>
        </try>
    </sequence>
</parallel>
```

So the above example is a lot to read at once, lets start slowly and look at first the rendezvous that was created in order to flag when the writer thread has completed so that the reader thread can detect this in its while loop and stop reading. The one thing to notice here is that even though we've used the **rendezvous_check** tag to validate that the other party has reached the rendezvous point we still need to visit this rendezvous point and we must use a **try/finally** model like we did here otherwise a failure in the code inside of that try could lead to an exception and the other thread would then sit waiting on the **rendezvous_visit** for eternity so the easiest way to solve this is to use the **try/finally** trick that was used here. Now the other more interesting part of the code is the use of the actual share point which was created called **MYSHARE**, our sets on the reader thread are done for every single write (but we could have opted to only do that ever X amount of writes to be more efficient).

On the reader thread side though we've used a **blocking share_get** call to make sure that we only start doing work once there is work to be done and that every time we go back to do a **share_get** we've received a brand new data from the writer thread. The only other thing to note here is that we chose to use a range to create a random range from which the reader thread can make requests, but we could have just set a property with the highest record number that can be retrieved from the server at this moment.

You can run the example this code that can be found at **tests/examples/communication/share1.xml**, but if you want to run it without the **Webdav** server setup then just comment out the **http_put** and **http_get** tags and you'll be able to just see the output of writing and reading certain record numbers.

A few other things to note about this example is that you can have multiple writers setting the record number on the same share point and that the two main sequence blocks can be on different machines running under a component tag and still this code would work the same way, of course at that point you'd want to make the number of **share_set** and **share_gets** to a more reasonable interval of every few hundred or thousand requests to make sure to not influence the amount of work your system can. Another thing to note if doing this in a distributed fashion is that you might want to not use a blocking **share_get** and instead use a rendezvous point to signal that there is at least 1 record written to the system before starting your readers and then have your readers check the if there are new records every few hundred or so iterations of reads.

Now within share points there are a few types you can choose from, currently here are the available types and their description. (the **type** can be specified at create time with the ***type*** attribute on the tag **share_create**)

Type	Description
single	The single share point type basically will hold a single instance of the last set done to this share. Any previous **share_set** is lost and can never be retrieved. This type is usually used when you just want to have share that contains the latest and greatest version of some information.
queue	The queue share point type basically will queue up all of the **share_set**s done and will return them in the same order back to the each of the share_gets that are done. If N threads are doing **share_get** on the same queue share point they will each get a different set from the queue and never will there be 2 threads have the same thing returned.
cumulative	The cumulative share point is special because just like the queue it will keep all the sets that were done in a queue, but when a **share_get** comes through this share point will return all of its current information back to that one **share_get**. So it acts

	like a buffer where you can write a bunch of different information and once one single **share_get** comes through it will be able to pick up all the information in a single request. You must be careful to not overload this type of share point with thousands of sets because you will break the agents with an out of memory exception.

Other share point types can be created and it is quite easy to implement a new one, have a look at the development guide for more information and an example on this.

Special things to know about Share and Rendezvous Points

There are a few things that should be noted when using share and rendezvous points in DTF, here is a list of somethings that may not always be obvious:

1. The component (be it runner or agent) where you create a share point or rendezvous point is where this element will reside. This means that the usage of this resource locally is as really fast, but other components have to go over the network to that component where the resource was created in order to use it. This is useful to know because you can choose to create certain shares or rendezvous points on agents that would geographically be close to each other in order to make a better use of this resource. In most cases both resources are just created on the runner and then shared across agents, but the option of creating the resource directly on the agents is available and should be used wisely.

2. Using a blocking **share_get** is sometimes required to make sure that you have at least a starting state before proceeding, but sometimes you can create an initial state by setting the right properties/ranges for your thread that is doing the **share_get** requests.

3. When using rendezvous points and checking if the other parties have reached the rendezvous point using the **rendezvous_check** you must make sure to visit once everyone else is there otherwise you'll leave the other parties hanging. In this same idea you have to be sure that you protect sensitive code that can fail to make sure to always visit the rendezvous point using a **try/finally** otherwise you will get threads stuck on a **rendezvous_visit**. Another way to avoid getting stuck is to use the ***timeout*** attribute on the **rendezvous_visit** that should be an amount of time after which something has to be seriously wrong with your test because it shouldn't have taken this long to reach this point.

4. The **share_set** tag can only accept a single tag to send, this was done on purpose so that you don't put a ton of logic into the underlying tag. In order to send a more complex group of actions you'll have to use a **function** and then use the **call** tag under neath the **share_set**. This allows for easier reading of code but it also keeps things like property resolution within the function call nice and clean.

Tricks and Tips

Dynamic Data

Within **DTF** sometimes you'll find yourself using external sources of data that aren't really dynamic because you just want to push this well known piece of data to a service and then get back a specific response that you know is certain for this type of data. Now this will work all fine until the day you're working on the next release of the product that now wants you to add a new field to your data such as the attribute *table_type* to all of the data of your tests. There are obviously always *n* ways of doing this but one of the simplest and elegant ways to do this is to make all of your static data contain a simple property for extra fields to be inserted at certain points in th object and set that property to nothing by default and then you can easily add your new fields at this point. Lets see how this works for a simple example (example test at **tests/examples/tips/dynamic_data.xml**). So we have a web service that accepts XML objects with the following format:

```
<operation>
    <insert table="X">
        <field name="A" value="V1"/>
        <field name="B" value="V2"/>
        <field name="C" value="V3"/>
        <field name="D" value="V3"/>
    </insert>
</operation>
```

So the above defines an operation on a table called X and this data is sent by various tests you wrote that need to create this test table before your tests can run. This all worked fine for version 1.0 of your product but now in version 1.1 the product will include a new field that you need to identify in each of your operations: insert, delete, etc. that includes a type attribute that identifies what type of a table is being created. The easiest thing to do to your original data is to add a few properties in the right place so that when you need to add information such as this for testing version 1.1 of the product you can simply load a different base properties file for each release. Here is how I'd change the previous data object to be more flexible for future releases:

```
<operation>
    <insert table="X" ${xml_insert_attr}>
        <field name="A" value="V1"/>
        <field name="B" value="V2"/>
        <field name="C" value="V3"/>
        <field name="D" value="V3"/>
        ${xml_insert_fields}
    </insert>
    ${xml_after_insert}
</operation>
```

I've added a few more properties in there that will allow this data to be more flexible in a future release. In your default v1.0 property file you'll have all of those properties set to nothing like so:

```
xml_insert_attr=
xml_insert_fields=
xml_after_insert=
```

While in your version 1.1 properties file you could have something more elaborate such as:

```
xml_insert_attr=table_type="testing"
xml_insert_fields=<field name="extra" value="somevalue"/>
xlm_after_insert=<commit/>
```

The exact same test can now easily use the same object with a small tweak and be able to run against both versions of the product. When running the test found at **tests/examples/tricks/dynamic_data.xml** you can easily specify the property ***version*** and change the output of the data being printed for the two currently supported versions of that data. Here's the example command line and output associated:

```
./ant.sh run_dtfx -Ddtf.xml.filename=tests/examples/tricks/dynamic_data.xml
...
     [java] INFO  08/10/2009 09:55:04 Log             - <operation>
     [java]     <insert table="X" >
     [java]         <field name="A" value="V1"/>
     [java]         <field name="B" value="V2"/>
     [java]         <field name="C" value="V3"/>
     [java]         <field name="D" value="V3"/>
     [java]
     [java]     </insert>
     [java]
     [java] </operation>
...

./ant.sh run_dtfx -Ddtf.xml.filename=tests/examples/tricks/dynamic_data.xml -Dversion=1.1
...
     [java] INFO  08/10/2009 10:00:56 Log             - <operation>
     [java]     <insert table="X" table_type="testing">
     [java]         <field name="A" value="V1"/>
     [java]         <field name="B" value="V2"/>
     [java]         <field name="C" value="V3"/>
     [java]         <field name="D" value="V3"/>
     [java]         <field name="extra" value="somevalue"/>
     [java]     </insert>
     [java]     <commit/>
     [java] </operation>
...
```

There you have the easiest way to keep your tests running between releases that can be easily extended into further releases by having your data dynamically altered by the properties you define during your execution. If you find that in order to debug issues its best to have a copy of the exact data object you were using then just make sure to output your data to an output file using the ***cat*** tag and then if something goes wrong you will always have the exact data used during that test run. In the example that exists in the **tests/examples/tricks** directory we also cover the usage of JSON data with an example similar to the one above.

Real World Scenarios

In this section we'll present a couple of real world testing scenarios that involve a real product that we'd like to test with DTF and walk through the various challenges of writing the tests.

Yahoo! Web Search API

The first product we'll test is just Yahoo's search engine using its Web Search REST API to do some pretty cool testing. More information on the Web Search feature can be found here http://developer.yahoo.com/search/web/V1/webSearch.html. So Yahoo has Web Search that can be used to do searches and the results from these searches are in XML. For example the following URL will search for the query "yahoo" on the web search service: http://api.search.yahoo.com/WebSearchService/V1/webSearch?appid=dtfdemo&query=yahoo . Now lets cover a few of the important testing areas involved with testing a product such as a web search API. We'll cover regression and FVT testing scenarios and walk through each of them and how to implement those tests correctly using DTF.

Regression Scenarios

Lets start with some simple tests that we can then create a simple regression test suite from:

1. We want to make sure that when we search for yahoo, the first result we get is a link to the www.yahoo.com web page.
2. We want to make sure that the number of results is always respected for numbers of results between 1 and 100
3. We want to make sure that no matter how many results are requested for the search string yahoo, we will always get back yahoo in first place.
4. Make a more generic test that can read query strings from a test file and the expected first result to do the previous tests all in one more advanced test.
5. A simple performance test that measures the performance of searching for the same string.
6. A test suite for regression testing that groups all of the previous tests and shows some of the test suite functionality available in DTF.

Now lets look at each of the previous test scenarios separately and construct each of the tests on their own to fully understand how to use DTF to your advantage. All of the tests that we'll cover in the following sections have the source code available at *tests/examples/websearch/regression* and can be easily executed from your DTF build.

Test Scenario 1

"We want to make sure that when we search for yahoo, the first result we get is a link to the www.yahoo.com web page."

So to talk to any web service from DTF we have a few HTTP tags that can be used to issue HTTP requests to any specified URI. Now for this specific test scenario we just want to do a HTTP GET and pass the URI encoded parameters *query* and *appid*. So here's what the HTTP request itself would look like in DTF:

```
<http_get uri="${websearch.uri}?appid=dtftest&amp;query=yahoo"/>
```

There's a small encoding in the URI that is necessary because you can't have certain characters such as &, < and > inside your XML. That is all we'd have to do in order to issue the HTTP GET that we need, now notice that the actual web search URI is hidden behind that property *websearch.uri*. That property is actually being created inside the default property file we're loading at the top of our test, here's the complete DTF XML file:

```
<script xmlns="http://dtf.org/v1" name="scenario1">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>test scenario 1 from the users's guide documentation</description>
    </info>

    <local>
        <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

        <loadproperties uri="storage://INPUT/websearch.properties"/>
    </local>

    <http_get uri="${websearch.uri}?appid=dtftest&amp;query=yahoo"/>
</script>
```

And here's the property file used:

```
websearch.host=http://api.search.yahoo.com
websearch.v1=WebSearchService/V1/webSearch
websearch.uri=${websearch.host}/${websearch.v1}
```

The usage of a property file like so allows us to easily overwrite the *websearch.host* from the command line and point our tests at a different test server. It also allows us to easily change the version of the API that we are testing. Being crafty like this with the amount of properties you put into your default property file is something that will make more sense once you start using the framework more and understanding the order in which properties are loaded and how to use this to your advantage.

The above test doesn't do any validation it just issues the HTTP GET and as long as the result is a 200 status code then it doesn't throw any exception and your test has passed. Now we wanted to validate that the very first result is in fact a reference to www.yahoo.com and not any other website. For this we first need to understand that most tags inside of DTF that do a determined action actually return an event back, this event can be easily used to get back the data from our tags execution or check for a successful completion of our tags work. Each tag has different event names with different attributes associated with each event, in order to figure out what the tag returns you should issue “ant doc” from your build workspace and this will build the DTF XML Documentation directly from the Java code. This documentation is extremely to find out the exact events that each tag throws. For this specific scenario the ***http_get*** returns an event with the name ***http.get*** which has the attributes: **status**, **statusmsg**, body, etc. For our specific purpose we want the response data that is inside of the body attribute and the status code so that we can make sure it was a successful HTTP response. So here's how we handle this in the actual XML code:

```
<property name="uri" value="${websearch.uri}?appid=dtftest&amp;query=yahoo&amp;results=1"/>
<log>HTTP Get on [${uri}]</log>
<http_get uri="${uri}"/>

<if>
    <eq op1="${http.get.status}" op2="200"/>
    <then>
        <log>${http.get.body}</log>
    </then>
    <else>
        <fail message="Failed http request got [${http.get.status}] status"/>
    </else>
</if>
```

In the previous example we've changed the test a bit to store the URI into a property so we can easily print it and see where we're trying to send the request. For those who are really paying attention you'll notice that we put another parameter into the ***http_get*** URI so that we'd limit the results to just 1 for the time being. Then we do our validation using the event that the ***http_get*** tag gives us. Now we're just printing the result and visually we can inspect the result and see that in fact www.yahoo.com was the first result. What we really want to do is to validate this result using the the test case itself. There is a very easy way of doing this by using the Xpath (read **XML Handling inside of DTF** section for more information) feature that allows you to apply XPath expressions directly on any property value. First lets look at the results of the body response so that we can craft the right XPath expression. Here's the response:

```
<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="urn:yahoo:srch"
    xsi:schemaLocation="urn:yahoo:srch
http://api.search.yahoo.com/WebSearchService/V1/WebSearchResponse.xsd"
    type="web"
    totalResultsAvailable="2147483647"
    totalResultsReturned="1"
    firstResultPosition="1"
    moreSearch="/WebSearchService/V1/webSearch?query=yahoo&amp;appid=dtftest&amp;region=us">
    <Result>
        <Title>Yahoo!</Title>
        <Summary>Yahoo! Internet portal provides email, news,
            shopping, web search, music, fantasy sports,and many other
            online products and services to consumers and businesses
            worldwide.
        </Summary>
        <Url>http://www.yahoo.com/</Url>
        <ClickUrl>http://uk.wrs.yahoo.com/_ylt=A0Je5VOZ3YlJB5EAaFXdmMwF;_ylu=X3oDMTB2cXVjNTM5BGN
vbG8DdwRsA1dTMQRwb3MDMQRzZWMDc3IEdnRpZAM-/SIG=119q91uph/EXP=1233858329/**http
%3A//www.yahoo.com/</ClickUrl>
        <DisplayUrl>www.yahoo.com/</DisplayUrl>
        <ModificationDate>1233734400</ModificationDate>
        <MimeType>text/html</MimeType>
   </Result>
</ResultSet>
```

In the above response we just want to confirm that the ***<URL>*** tag has the http://www.yahoo.com/ value and that entry is the first result in the ResultSet returned from the service. So with the an XPath expression such as **`//ResultSet/Result[1]/Url/text()`** would give us the exact URL of the first link returned. Here's how we'd do this in the test itself:

```
<script xmlns="http://dtf.org/v1" name="scenario1">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>test scenario 1 from the users's guide documentation</description>
    </info>

    <local>
        <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

        <loadproperties uri="storage://INPUT/websearch.properties"/>
    </local>

    <property name="uri" value="${websearch.uri}?appid=dtftest&amp;query=yahoo&amp;results=1"/>
    <log>HTTP Get on [${uri}]</log>
    <http_get uri="${uri}"/>

    <!-- check the status code was successful -->
    <if>
        <neq op1="${http.get.status}" op2="200"/>
        <then>
            <fail message="Failed http request got [${http.get.status}] status"/>
        </then>
    </if>

    <!--  check that the first result in the ResultSet is www.yahoo.com -->
    <property name="firstresult.url"
              value="${http.get.body:xpath:/ResultSet/Result[1]/Url/text()}"/>
    <if>
        <neq op1="${firstresult.url}" op2="http://www.yahoo.com/"/>
        <then>
            <fail message="Result was not http://www.yahoo.com] got [${firstresult.url}]."/>
        </then>
    </if>
</script>
```

So there's our first scenario covered and with just 40 lines of XML code we are able to do an HTTP request and validate that it is returning the right data along with a successful HTTP response.

Test Scenario 2

“We want to make sure that the number of results is always respected for numbers of results between 1 and 100.”

Our new scenario has the requirement of counting the number of results that come back in the response from the web service. Now this can easily be done by using an **XPath** function called count which allows us to know exactly how many XML nodes are at a certain level in an XML document. The other thing that this scenario requires is changing the number of results that we're accepting on the return for each iteration and for this we'll use the ***for*** loop tag that **DTF** has. Here's one possible solution:

```
<script xmlns="http://dtf.org/v1" name="scenario1">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>test scenario 2 from the users's guide documentation</description>
    </info>

    <local>
        <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

        <loadproperties uri="storage://INPUT/websearch.properties"/>
    </local>

    <for property="count" range="1..100">
           <http_get uri="${websearch.uri}?appid=dtftest&amp;query=yahoo&amp;results=$
{count}"/>

           <!-- check the status code was successful -->
           <if>
               <neq op1="${http.get.status}" op2="200"/>
               <then>
                   <fail message="Failed http request got [${http.get.status}] status"/>
               </then>
           </if>

        <log>Checking ResultSet for ${count} results</log>
           <!--  check that the first result in the ResultSet is www.yahoo.com -->
           <property name="result.count"
                     value="${http.get.body:xpath:count(/ResultSet/Result)}"
                 overwrite="true"/>
           <if>
               <neq op1="${result.count}" op2="${count}"/>
               <then>
                   <fail message="Expected ${count} results but got ${result.count}"/>
               </then>
           </if>
    </for>
</script>
```

The previous example just has a for loop and now validates that the number of results is always equal to the exact result count we requested when doing the search. This test takes about a minute to execute because we're doing each of the requests sequentially, if we wanted to parallelize the work a bit we could use a ***distribute*** tag to easily define the maximum amount of concurrency to use during the test and efficiently use X number of threads to finish our task faster. The concurrent implementation of the previous test can be found at ***tests/examples/websearch/scenario2_concurrent.xml*** and it completes in half the time as the previous one using up to 5 threads.

Test Scenario 3

> “We want to make sure that no matter how many results are requested for the search string yahoo, we will always get back yahoo in first place.”

With this scenario we start to see that we have the necessity to validate that a certain ***ResultSet*** has the first entry equal to X and that we have to repeat some of the XML code we already have above. Here is where we should start to notice that the use of functions would be welcome. So first lets put together a function that can process a response that we pass as an argument and validate that the first result is equal to another parameter we give it, as well as making sure that the result count is the expected count. Here's an example function:

```
        <function name="validate_response">
            <param name="resultset" type="required"/>
            <param name="count" type="required"/>
            <param name="firsturl" default="NONE"/>

            <!-- check the resultset count -->
                <property name="result.count"
                          value="${http.get.body:xpath:count(/ResultSet/Result)}"
                      overwrite="true"/>
                <if>
                    <neq op1="${result.count}" op2="${count}"/>
                    <then>
                        <fail message="Expected ${count} results but got ${result.count}"/>
                    </then>
                </if>

            <if>
                <neq op1="firsturl" op2="NONE"/>
                <then>
                              <!--  check that the first result in the ResultSet is ${firsturl} -->
                              <property name="firstresult.url"
                                        value="$
    {http.get.body:xpath:/ResultSet/Result[1]/Url/text()}"/>
                              <if>
                                  <neq op1="${firstresult.url}" op2="${firsturl}"/>
                                  <then>
                                      <fail message="Result was not [${firsturl}] got [$
    {firstresult.url}]."/>
                                  </then>
                              </if>
                </then>
            </if>
        </function>
```

The function defined here has 2 required parameters that includes the ***ResultSet*** itself along with the expected result count. Then there's the optional argument *firsturl* which when present will result in checking the first URL of the ***ResultSet*** for that exact value. With this function we can now write our test without having to repeat the same XML code inside this test as well and instead have a call to this function, here's the example:

```
    <script xmlns="http://dtf.org/v1" name="scenario3">
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rlgomes@yahoo-inc.com</email>
            </author>
            <description>test scenario 3 from the users's guide documentation</description>
        </info>
        <local>
            <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
            <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

            <loadproperties uri="storage://INPUT/websearch.properties"/>

            <import uri="storage://INPUT/util.xml"/>
        </local>

        <for property="count" range="1..10">
            <log>Iteration ${count}</log>
               <http_get uri="${websearch.uri}?appid=dtftest&amp;query=yahoo&amp;results=${count}"
                      onFailure="fail"/>
            <call function="validate_response">
                <property name="resultset" value="${http.get.body}"/>
                <property name="firsturl" value="http://www.yahoo.com/"/>
                <property name="count" value="${count}"/>
            </call>
        </for>
    </script>
```

The above test is actually smaller than our original simple test that just validated doing a search query for the string yahoo. This is the power of using functions, once you have the right functions defined your tests become simpler and cleaner and end up doing so much more. Notice I used the property *onFailure="fail"* on the **http_get** so that if there is a non 200 response the test will automatically fail. This property is used to control when the tag will throw an exception or just return the event and allow you to make a decision in the case of a failure.

Test Scenario 4

"Make a more generic test that can read query strings from a test file and the expected first result to do the previous tests all in one more advanced test."

Now we want to make a test that is data driven, in our case we'll make use of an XML file that contains the search query and the expected first result that we want for that specific query and use this to drive our scenario4.xml test. Here's how the input XML file may look like:

```
<cases>
    <case><query>yahoo</query><firsturl>http://www.yahoo.com/</firsturl></case>
    <case><query>google</query><firsturl>http://www.google.com/</firsturl></case>
    <case><query>finance</query><firsturl>http://finance.yahoo.com/</firsturl></case>
    <case><query>mail</query><firsturl>http://mail.yahoo.com/</firsturl></case>
</cases>
```

The previous data file contains a very simple structure that can identify the query string and the expected first result. Now to process this inside of DTF and make a data driven test is pretty straightforward and includes using all of the features shown up till now for this real world scenario, here's what it may look like and a little explanation on some of the elements:

```
<script xmlns="http://dtf.org/v1" name="scenario3">
    <info>
        <author>
            <name>Rodney Gomes</name>
            <email>rlgomes@yahoo-inc.com</email>
        </author>
        <description>test scenario 3 from the users's guide documentation</description>
    </info>
```

```
        <local>
            <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
            <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

            <loadproperties uri="storage://INPUT/websearch.properties"/>

            <import uri="storage://INPUT/util.xml"/>

            <property name="data" uri="storage://INPUT/scenario4_data.xml"/>
        </local>

        <for property="case" range="xpath(${data},/cases/case)">
            <property name="query" value="${case:xpath:/case/query}" overwrite="true"/>
            <property name="expected.url" value="${case:xpath:/case/firsturl}" overwrite="true"/>
            <log>Query: ${query} Expected First Result: ${expected.url}</log>

               <http_get uri="${websearch.uri}?appid=dtftest&amp;query=${query}&amp;results=5"
                         onFailure="fail"/>
            <call function="validate_response">
                <property name="resultset" value="${http.get.body}"/>
                <property name="firsturl" value="${expected.url}"/>
                <property name="count" value="5"/>
            </call>
        </for>
    </script>
```

In this example we load the data XML file into a property that we can then use to loop through each of the ***<case>*** tags inside of the cases and further decompose those with XPath expressions to test them for the same validity we already did in the previous scenario. Again notice this test is less than 40 lines long and you can easily increase the amount of testing it covers by changing the data it is using. This data could even be in separate files that are fed as a property to this test and used in different scenarios.

Test Scenario 5

"A simple performance test that measures the performance of searching for the same string."

This scenario wants us to measure the performance of our searches. Now DTF by default has already measured each and every event that is thrown. So in this case our ***http_get*** already have the necessary measurements for us to be able to get our performance numbers. The only thing we need to do is to tell the system to record these events and then process these events with the ***query*** and ***stats*** tags. Lets start with *scenario3.xml* and use this to build the performance measurement and reporting code around it. Here's the *scenario5.xml* file:

```
    <script xmlns="http://dtf.org/v1" name="scenario5">
        ...

        <record uri="storage://OUTPUT/websearch_events.txt">
               <for property="count" range="1..10">
                   <log>Iteration ${count}</log>
                      <http_get uri="${websearch.uri}?appid=dtftest&amp;query=yahoo"
                             onFailure="continue"/>
               </for>
        </record>

        <!-- we only want the events that were successful -->
        <query uri="storage://OUTPUT/websearch_events.txt"
               cursor="websearch"
               event="http.get">
            <where>
                <eq op1="status" op2="200"/>
            </where>
        </query>

        <stats cursor="websearch" event="perf"/>

        <log>
            Yahoo! Web Search Performance
            *****************************
            Avg Search Time: ${perf.avg_dur}ms
            Max Search Time: ${perf.max_dur}ms
            Min Search Time: ${perf.min_dur}ms
        </log>
    </script>
```

As you can see the previous script just wrapped the loop around our ***http_get*** with a simple ***record*** tag and the events are being recorded to that file. These events measure the exact time it took to do the HTTP request, so if you have any extra logic you'd like to put in there you're only limiting the amount of requests per second you can do and not interfering with the real latency measurements. Then using the ***query*** tag we identify which requests we want to calculate statistics on. In this case we narrowed the events to those who had a *status* of 200. Using the ***stats*** tag we easily get the maximum, minimum, average latencies, along with some other measurements that we won't cover here.

Test Scenario 6

> "A test suite that groups all of the previous tests and shows some of the test suite functionality available in DTF."

So now lets create a test suite and call all of our existing scenarios. This is easily achieved using the ***testsuite*** tag to identify the name of our test suite and then calling all of our test cases using the ***testscript*** tag to execute the other scenarios. While the test suite executes all of the test scripts we specify it will not terminate the execution of the test suite just because one test script has failed, instead it will continue to execute until all of them have been executed and at the end of the test suite execution it will report failures or successes. To record these results we have to use the tag ***results*** to push the results of our test suite somewhere, for this test we'll push them to the console, but there are a few other options which include an external results file in XML that can be post processed for uploading to a test case results management software. Here's the current implementation:

```
    <script xmlns="http://dtf.org/v1" name="tessuite">
        <info>
            <author>
                <name>Rodney Gomes</name>
                <email>rlgomes@yahoo-inc.com</email>
            </author>
            <description>test suite from the users's guide documentation</description>
        </info>

        <local>
            <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
            <createstorage id="TINPUT" path="${dtf.xml.path}"/>
            <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

            <loadproperties uri="storage://INPUT/websearch.properties"/>
        </local>

        <result type="console">
               <testsuite name="websearch_testsuite">
                   <testscript uri="storage://TINPUT/scenario1.xml"/>
                <!--  avoiding the slower implementation of scenario 2 -->
                   <!--  <testscript uri="storage://TINPUT/scenario2.xml"/> -->
                   <testscript uri="storage://TINPUT/scenario2_concurrent.xml"/>

                   <testscript uri="storage://TINPUT/scenario3.xml"/>
                   <testscript uri="storage://TINPUT/scenario4.xml"/>
                   <testscript uri="storage://TINPUT/scenario5.xml"/>
               </testsuite>
        </result>
    </script>
```

The output from the console result looks something like this:

```
[java] INFO  04/02/2009 14:01:16 ConsoleResults  - Testsuite: websearch_testsuite passed. 5 passed testcases.
```

And it simply states that you executed 5 test cases and all of them passed. If there were failures it would report how many and you'd have to look through the logs to find the stack traces. The use of the XML result type is preferred because it saves the exact error per test case making it easier to figure out what failed and where.

FVT Scenarios

Now that we've ran through those tests lets create more functional tests that can actually validate the correctness of a certain feature in the web search API. Here are a few test scenarios we'd like to cover:

1. Validate when doing the same query from multiple machines and multiple threads at the same time that the results are always the same within a small window of time (lets say about 30s). We'll need to figure out how similar the results need to be and how much difference we're willing to tolerate in this scenario.
2. Validate that all of that the results that state that they have cached copies of the pages, always have the cached copy available and that you can at least retrieve it. Would be even better to validate that the cached copy is similar to the real page itself, if this real page is available at the moment.
3. Testing that commonly mis-joined words or words in different orders yield similar results from the web search API. Headache, head-ache, head ache, etc.

Test Scenario 1

“Validate when doing the same query from multiple machines and multiple threads at the same time that the results are always the same within a small window of time (lets say about 30s). We'll need to figure out how similar the results need to be and how much difference we're willing to tolerate in this scenario.”

So first we need to see how similar the results themselves have to be within a small window of time. For this purpose we'd need to know the inner workings of the web search API itself and since I personally don't have that I'll try issuing some searches and saving the results to see exactly how the results change from query to query over a few seconds. After doing this a few times I've found there is some expiration and signature parameters in the click URL and cache URL. We will have to be careful when comparing these results because of these fields that have slight changes. Lets start with designing a test for this scenario that can operate on anywhere from 1 to N clients at the same time and issue the same query on all machines at the same time using up to X threads. Here's the first version of the test:

```
<script xmlns="http://dtf.org/v1" name="scenario1">
    ...

    <local>
        <createstorage id="INPUT" path="${dtf.xml.path}/input"/>
        <createstorage id="OUTPUT" path="${dtf.xml.path}/output"/>

        <loadproperties uri="storage://INPUT/websearch.properties"/>

        <property name="websearch.load_timer" value="30s"/>
        <property name="websearch.workers" value="2"/>
    </local>

    <!-- lock the number of clients we'll need for this test -->
    <for property="client" range="1..${websearch.clients}">
        <local>
            <lockcomponent id="CLIENT${client}"/>
        </local>
    </for>

    <property name="uri"
              value="${websearch.uri}?appid=dtftest&amp;query=yahoo"/>

    <record uri="storage://OUTPUT/scenario1_search_events.txt">
           <parallelloop property="client" range="1..${websearch.clients}">
               <component id="CLIENT${client}">
                   <distribute id="mydist"
                               workers="${websearch.workers}"
                               timer="${websearch.load_timer}">
                                 <http_get uri="${uri}"/>
                   </distribute>
               </component>
           </parallelloop>
    </record>
</script>
```

The first version we've found a way of dynamically locking components based on a simple *websearch.clients* property that identifies how many clients we want to use for this test. We've also are recording the results from each of the requests that was issued, so the only thing missing is how do we take those results and really compare them to each other ? Well one idea is to pick up the first result and then compare it to all the other results in the same events file. We'd need to then do some smart comparison but right now lets just leave that to a secondary function that will define when two results are similar enough for our needs. Here's the piece of XML needed to run through the results and compare them as we previously explained:

```
    <query uri="storage://OUTPUT/scenario1_search_events.txt"
           cursor="results">
        <where>
            <!-- we only want the successful operations -->
            <eq op1="status" op2="200"/>
        </where>
    </query>

    <nextresult cursor="results"/>
    <property name="firstresult" value="${results.body}"/>

    <iterate cursor="results">
        <call function="compareResults">
            <property name="first" value="${firstresult}"/>
            <property name="other" value="${results.body}"/>
        </call>
    </iterate>
```

```
<!-- this function throws an exception when the data is not similar
     enough for our requirements -->
<function name="compareResults">
    <param name="first" type="required"/>
    <param name="other" type="required"/>

    <log>Checked ${other}</log>
</function>
```

We're now using the *query* feature of DTF to read back the events we previously recorded with the *record* tag. This allows us to run through those results either by using the *nextresult* tag or using the more efficient *iterate* tag to iterate through the existing values. First thing we'll do is make the function validate that all of the same results appear in both **ResultSets** by comparing the titles and URLs for each result. Here's our first approach at tackling this similarity function:

```
<function name="compareResults">
    <param name="first" type="required"/>
    <param name="other" type="required"/>

    <for property="result"
         range="xpath(${first},/ResultSet/Result)">
        <property name="title"
                  value="${result:xpath:/Result/Title/text()}"
                  overwrite="true"/>
        <property name="url"
                  value="${result:xpath:/Result/Url/text()}"
                  overwrite="true"/>

        <property name="lookup_result"
                  value="${other:xpath:/ResultSet/Result[Title[text() = '${title}']]}"
                  overwrite="true"/>

        <if>
            <eq op1="${lookup_result}" op2=""/>
            <then>
                <fail message="No result found for ${title} in the other ResultSet."/>
            </then>
        </if>

        <property name="lookup_title"
                  value="${lookup_result:xpath:/Result/Title/text()}"
                  overwrite="true"/>
        <property name="lookup_url"
                  value="${lookup_result:xpath:/Result/Url/text()}"
                  overwrite="true"/>
        <if>
            <neq op1="${title}" op2="${lookup_title}"/>
            <then>
                <fail message="Didn't find the result ${result} in the other ResultSet"/>
            </then>
        </if>

        <if>
            <neq op1="${url}" op2="${lookup_url}"/>
            <then>
                <fail message="Didn't find the result ${result} in the other ResultSet"/>
            </then>
        </if>
    </for>
</function>
```

There's a pretty hefty function that is doing some pretty complex XML operations to compare to XML results. I've removed the comments from the original code so that we could fit just the XML code here for analysis purposes. Now what we've constructed is a simple way of iterating through each **Result** in the **ResultSet** of the first yahoo query and we make sure the same title and URL show up in a Result of any other search query we did from any other request. If we don't find one of the results then we throw an exception and break the test because we have missing results. This obviously may miss the case where some of the other **ResultSets** have additional **Results** that aren't in the first query, but this isn't the case since all **ResultSets** have the same size, so we can't have any **Results** in the other **ResultSets** that isn't already part of the first (nice optimization for the time being). There are a additional things we could check for, but I'm not going to make this function more complex that it needs to be for the documentation purposes. The full testcase can be found at *tests/examples/websearch/fvt/scenario1.xml*.

So now this specific test can be controlled from the command line, these properties include:

- *websearch.workers* – defines the number of threads to use on each client machine.
- *websearch.clients* – defines the number of DTFA's to use during the test run.
- *websearch.load_timer* – identifies the amount of time to run the test for.

For example if we wanted to run this test for 1h from 3 clients with 20 threads on each client doing as much work as posible then we'd issue the command line:

```
ant.sh run_dtfx -Ddtf.xml.filename=tests/examples/websearch/fvt/scenario1.xml
-Dwebsearch.load_timer=1h -Dwebsearch.clients=3 -Dwebsearch.workers=20
```

Now one thing to note is that this test assumes that from the time it took to do the first request till the time runs out and an hour has gone by, we assume that all of the results will contain the same amount of data and no major changes may have occurred. Now if we knew that in some cases data would change and that only about 80% of the results would be similar, then we could code this into our *compareResults* function by counting the number of times we have found and not found a match per result and then doing some math on that calculate if at least 80% of the results were similar. The sky is the limit really when it comes to the amount of validation you want to do for the results.

Test Scenario 2

> “Validate that all of that the results that state that they have cached copies of the pages, always have the cached copy available and that you can at least retrieve it. Would be even better to validate that the cached copy is similar to the real page itself, if this real page is available at the moment.”

So this scenario wants to test the availability of cached pages and that they are somewhat fresh in relation to the origin page. Now for our simple example we'll just extract the cache link from the **Results** and validate that the page exists. Of course we're going to do this from multiple machines in a multi threaded fashion again because we really want to stress the system while we're testing the same functionality on multiple different queries.

The solution we've reached makes use of some new functionality which is the ability to use functions on the component side. This feature is enabled by just setting the attribute ***export=”true”*** on the ***function*** declaration. The test itself is very similar to previous ones and it does the validation while the activities are going on and counts up the number of cached/noncached results and spits that out to the logs for visual inspection. There are more things that could be done here such as validating that the cached page is somewhat recent but in order to know exactly how fresh cached pages need to be in order to meet the designed values. For this simple demonstration purpose we just wanted to show how little test cases change and how simple it is to read through a test and understand what it does. To see the test have a look at *tests/examples/websearch/fvt/scenario2.xml.*

Test Scenario 3

> “Testing that commonly mis-joined words or words in different orders yield similar results from the web search API. Headache, head-ache, head ache, etc.”

At this point the next test scenario is actualy just a simple permutation of some of the existing tests. So we want to be able to run multiple queries for a list of mis-joined words that should yield the same results. Lets define the group of mis-joined words using some ranges and then lets define a comparison function that can easily validate that the results are at least 80% similar, between the various spelling of the same word.

Lets start with defining the list of words that should yield similar results:

- headache, head-ache, head ache
- yahoo, Yahoo, YaHoo, YaHOO!

Now lets keep that small until we've designed the test and then we can keep adding more data to process without having to make changes to the test itself. So after some testing we can easily see that results for those headache searches are within about 40% of similarity while the yahoo searches are 100% the same results because its just a different spelling of the same search string. Then I added a few other search queries and made the test dynamic in the sense that for each search query it can tolerate a different % of similarity between the searches. The test itself can be found at *tests/examples/websearch/fvt/scenario3.xml*. We'll have a look at the main loop in the test just because it uses some new neat tricks to validate the searches from one run to the next.

```
<for property="words" range="${websearch.similar.words}">
    <property name="last" value="" overwrite="true" />

    <property name="word.tolerance"
              value="${websearch.similar.words.${words}.tolerance}"
              overwrite="true"/>
    <log>
         Queries [${websearch.similar.words.${words}}]
         with tolerance of ${word.tolerance}
    </log>

    <for property="word"
         range="${websearch.similar.words.${words}}">

        <!-- url encode for safety, because spaces and any
             special characters need to be encoded -->
        <urlencode source="${word}" result="word"/>
        <http_get uri="${uri}&amp;query=${word}" />

        <if>
            <neq op1="${last}" op2="" />
            <then>
                <call function="verifySimilarity">
                    <property name="prev" value="${last}" />
```

```
                        <property name="next" value="${http.get.body}" />
                        <property name="tolerance"
                            value="${word.tolerance}" />
                    </call>
                </then>
            </if>

            <property name="last"
                      value="${http.get.body}"
                      overwrite="true" />
        </for>
    </for>
```

In this loop we use the variable *${last}* to figure out if we have a previous search that we can actually compare with and if we don't (on the very first search) then we just skip the comparison part and make sure to set *${last}* to the last search you did. The other thing to notice in this loop is how we use the words that come from *${websearch.similiar.words}* to pick the correct *${websearch.similar.words.XXX}* property that contains the similar words that should have the similar search results.