IBM® Tivoli® Software

IBM Tivoli Workload Scheduler
V8.5.1 engine Capacity Planning

Document version 1.1

Leonardo Lanni
Monica Rossi
Tivoli Workload Automation Performance Team - IBM Rome Lab

IBM
Executive summary

Tivoli Workload Scheduler V8.5.1 engine capacity plan test activity is composed of a set of test scenarios that were run against the Tivoli Workload Scheduler engine server upon 5 distinct HW configurations with increasing power (in terms of available HW resources), each time providing 3 different workloads (light, medium, and heavy, in terms of scheduling objects), for a total of 15 combinations.

Test scenarios have the purpose of providing details about the Tivoli Workload Scheduler engine server resource usage while they run, for each of the input workloads and server configuration.

For each configuration, these are the 4 scenarios that ran:

1. Plan preparation: prepare the plan composed of the scheduling objects;
2. Plan execution: run the plan composed of the scheduling objects with all the constraints and dependencies;
3. Plan extension: extend the current plan;
4. Dynamic job submission: submit jobs to the engine:
   - When no plan is running
   - When the plan is running.

Tests were run in a completely virtualized environment, also using 32 Tivoli Workload Scheduler fault-tolerant agent instances installed on 8 distinct workstations, with the purpose of running the produced plan.

Monitoring tools were used to record Tivoli Workload Scheduler engine server resource usage, for each test run.

Collected data was analyzed and developed, to provide the reader with information about correct configurations, settings, and tunings to obtain suitable performances for each test.

Test scenario descriptions, analysis, results, and recommendations are given in this document, to guide the reader in the process of building an appropriate scheduling environment for its own input workload.

Each test from light to heavy was successfully run and completed on each of the 5 Tivoli Workload Scheduler engine server configurations, both from a functional and a performance/scalability point of view, reaching the following result for the upper scaling configuration:

1. Plan preparation: 3 plans up to 300,000 jobs, packed in 15,000 job streams, containing 300,000 dependencies and constraints
2. Plan execution: 3 plans up to 300,000 jobs, packed in 15,000 job streams, containing 300,000 dependencies and constraints
3. Plan extension: light workload plan extended 3 times
4. **Dynamic job submission: all jobs successfully submitted.**

Collected test results, in terms of monitored Tivoli Workload Scheduler engine server resource usage, were analyzed and developed to provide useful information to get the best performances for the scheduling environment to be built.
# CONTENTS

Executive summary ......................................................................................................... ii3

Contents ............................................................................................................................ v

List of Figures ................................................................................................................ ..viii

List of Tables ................................................................................................................... ix

Revision History ............................................................................................................. x

1  Overview ................................................................................................................ 1

   1.1 The Tivoli Workload Scheduler network ............................................................ 1

   1.2 Manager and agent types ................................................................................ 2

   1.3 Topology ......................................................................................................... 2

   1.4 Tivoli Workload Scheduler scheduling objects .............................................. 3

   1.5 Graphical view of Tivoli Workload Scheduler components involved in the capacity planning test activities ................................................................. 6

2  Goal ......................................................................................................................... 6

3  Approach .................................................................................................................. 8

   3.1 Installation and setup ....................................................................................... 8

   3.2 Test initial conditions ...................................................................................... 9

   3.3 Database information and tuning settings ...................................................... 9

4  Environment ............................................................................................................ 10
# IBM Tivoli Workload Scheduler V8.5.1 engine Capacity Planning

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Tivoli Workload Scheduler V8.5.1 engine</td>
<td>10</td>
</tr>
<tr>
<td>4.1.1 Tivoli Workload Scheduler engine embedded WebSphere Application Server heap size configuration and tuning</td>
<td>11</td>
</tr>
<tr>
<td>4.2 DB2 Database</td>
<td>12</td>
</tr>
<tr>
<td>4.3 Tivoli Workload Scheduler V8.5.1 fault-tolerant agents</td>
<td>13</td>
</tr>
<tr>
<td>4.3.1 Fault-tolerant agent full status configuration</td>
<td>14</td>
</tr>
<tr>
<td>4.3.2 Mailman servers and message queues</td>
<td>15</td>
</tr>
<tr>
<td>4.4 Network</td>
<td>15</td>
</tr>
<tr>
<td>5 Scenarios</td>
<td>16</td>
</tr>
<tr>
<td>5.1 Engine setup and database population</td>
<td>17</td>
</tr>
<tr>
<td>5.2 Plan generation (JnextPlan)</td>
<td>18</td>
</tr>
<tr>
<td>5.3 Plan execution and static scheduling (Execution)</td>
<td>19</td>
</tr>
<tr>
<td>5.4 Plan extension and further tests (Extension)</td>
<td>22</td>
</tr>
<tr>
<td>5.5 Dynamic job submission (Submit)</td>
<td>25</td>
</tr>
<tr>
<td>6 Workload</td>
<td>27</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>27</td>
</tr>
<tr>
<td>6.2 Workloads for plan preparation, plan execution, and JnextPlan further test and extensions</td>
<td>27</td>
</tr>
<tr>
<td>6.2.1 Jobs categories</td>
<td>28</td>
</tr>
<tr>
<td>6.2.2 Jobs execution time</td>
<td>30</td>
</tr>
<tr>
<td>6.2.3 Job recovery policy</td>
<td>31</td>
</tr>
<tr>
<td>6.2.4 Job priority</td>
<td>31</td>
</tr>
<tr>
<td>6.2.5 Job dependencies and topology</td>
<td>31</td>
</tr>
<tr>
<td>6.2.6 Prompt dependencies</td>
<td>32</td>
</tr>
<tr>
<td>6.2.7 File dependencies</td>
<td>32</td>
</tr>
</tbody>
</table>
6.2.8 Resource dependencies ................................................................. 33
6.2.9 Light workload topology and dependencies ................................. 34
6.2.10 Medium workload topology and dependencies ........................... 36
6.2.11 Heavy workload topology and dependencies ............................... 38
6.2.12 Job stream properties: “AT” ....................................................... 40
6.2.13 Job stream properties: carryforward .......................................... 40
6.2.14 Job stream properties: priority .................................................. 40
6.2.15 Job stream properties: limit ........................................................ 40
6.2.16 Job stream properties: dependencies and topology .................... 40

6.3 Dynamic job submission workload .................................................. 42

6.3.1 Plan scheduling objects ............................................................... 42
6.3.2 Job submission scheduling objects .............................................. 44

7 Conclusions and recommendations .................................................... 46

7.1 JnextPlan – Recommendations ....................................................... 46
7.2 Execution – Recommendations ....................................................... 47
7.3 Extension – Recommendations ....................................................... 48
7.4 Dynamic – Recommendations ....................................................... 49
7.5 Other suggestions and tuning ......................................................... 49
LIST OF FIGURES

Figure 1 - Tivoli Workload Scheduler network example ............................................................... 1
Figure 2 - Schema of the Tivoli Workload Scheduler components involved in the capacity planning
           test activities .......................................................................................................................... 6
Figure 3 - DB2 tunings .................................................................................................................. 13
Figure 4 - Tivoli Workload Scheduler 8.5.1 engine Capacity Plan environment description ....... 16
Figure 5 - Process to automate the input generation for scheduling object creation in TWS
           database, to generate the workload .................................................................................. 18
Figure 6 - Capacity Plan JnextPlan and Execution phases details .............................................. 22
Figure 7 - JnextPlan further tests and extension details ........................................................... 24
Figure 8 - JnextPlan extension part details ................................................................................ 24
Figure 9 - Dynamic job submission test details ........................................................................ 26
Figure 10 - Workload balancing between Master and fault-tolerant agents ............................... 28
Figure 11 - Heavy job script file ................................................................................................ 29
Figure 12 - Job category percentages over the total number of jobs run in a given workload ...... 30
Figure 13 - Light workload: job dependencies graphical description for a generic job stream named
            x ....................................................................................................................................... 35
Figure 14 - Medium workload: jobs dependency graphical description for a generic job stream named
            x ....................................................................................................................................... 37
Figure 15 - Heavy workload: jobs dependency graphical description for a generic job stream named
            x ....................................................................................................................................... 39
Figure 16 - Job stream *follows* dependencies in a given fault-tolerant agent instance on a specific
            fault-tolerant agent_i machine ....................................................................................... 41
Figure 17 - Dynamic job submission workload description .......................................................... 43
Figure 18 - Dynamic job submission: plan details ..................................................................... 44
Figure 19 - Dynamic job submission: jobs to submit details ....................................................... 45
LIST OF TABLES

Table 1 - Capacity Plan test names by configuration and workload………………………………………8
Table 2 - Tivoli Workload Scheduler 8.5.1 engine configuration descriptions…………………………10
Table 3 - Workload configuration and full status related setting………………………………………15
Table 4 - Number of jobs, job streams, and dependencies per workload ………………………………27
Table 5 - Jobs category details ……………………………………………………………………………29
Table 6 - Job prompt dependencies details for any job stream named x……………………………32
Table 7 - Job file dependency details for any job stream named x ……………………………………33
Table 8 - Resource dependencies for jobs of generic job streams named x ……………………………33
Table 9 - Light workload details …………………………………………………………………………34
Table 10 - Resource types and amounts for Light workload …………………………………………34
Table 11 – Light workload: job follows dependency topology for a given job streams named x……35
Table 12 - Medium workload details ……………………………………………………………………36
Table 13 - Resource types and amounts for Medium workload ………………………………………36
Table 14 – Medium workload: job follows dependency topology for a given job stream named x..37
Table 15 - Heavy workload details ………………………………………………………………………38
Table 16 - Resource types and amounts for Heavy workload …………………………………………38
Table 17 - Heavy workload: job follows dependency topology for a given job streams named x...39
Table 18 - JnextPlan recommendations …………………………………………………………………47
Table 19 - Execution - recommendations for best performances ………………………………………48
Table 20 - Extension - Recommendations for best performances on several features ………………48
Table 21 - Dynamic job submission, recommendations …………………………………………………49
### REVISION HISTORY

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revised By</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>09/09/2011</td>
<td>0.1</td>
<td>L. Lanni</td>
<td></td>
</tr>
<tr>
<td>26/09/2011</td>
<td>1.0</td>
<td>M. Rossi</td>
<td></td>
</tr>
<tr>
<td>17/10/2011</td>
<td>1.1</td>
<td>M. Rossi</td>
<td></td>
</tr>
</tbody>
</table>
1 Overview

1.1 The Tivoli Workload Scheduler network

A Tivoli Workload Scheduler network is made up of the workstations, or CPUs, on which jobs and job streams are run.

A Tivoli Workload Scheduler network contains at least one Tivoli Workload Scheduler domain, the master domain, in which the master domain manager is the management hub.

Additional domains can be used to divide a widely distributed network into smaller, locally managed groups.

![Figure 1 - Tivoli Workload Scheduler network example](image)

Using multiple domains reduces the amount of network traffic by reducing the amount of communication required between the master domain manager and other computers.

In a single domain configuration, the master domain manager maintains communication with all the workstations in the scheduling network.

In a multi-domain configuration, the master domain manager communicates with the workstations in its domain and with the subordinate domain managers.

The subordinate domain managers, in turn, communicate with the workstations in their domains and with their subordinate domain managers.

Multiple domains also provide fault-tolerance by limiting the problems caused by losing a domain manager to a single domain.
To limit the effects further, it is possible to designate backup domain managers to take over if their domain managers fail.

Every time the production plan is created or extended, the master domain manager creates a production control file, named Symphony.

Tivoli Workload Scheduler is then restarted in the network, and the master domain manager sends a copy of the new production control file to each of its automatically-linked agents and subordinate domain managers.

The domain managers, in turn, send copies to their automatically-linked agents and subordinate domain managers.

After the network is started, scheduling messages such as job starts and completions are passed from the agents to their domain managers, through the parent domain managers to the master domain manager.

The master domain manager then broadcasts the messages throughout the hierarchical tree to update the production control files of domain managers and fault-tolerant agents running in Full Status mode.

### 1.2 Manager and agent types

Primarily, workstation definitions refer to physical workstations. However, in the case of extended and network agents, the workstations are logical definitions that must be hosted by a physical Tivoli Workload Scheduler workstation.

Tivoli Workload Scheduler workstations can be of the following types:

- **Master domain manager (MDM)**

  The domain manager is the topmost domain of a Tivoli Workload Scheduler network. It either contains or connects to the relational database that stores the scheduling object definitions. It creates or updates the production file when the plan is created or extended and distributes it in the network. It performs all logging and reporting for the network.

- **Fault-tolerant agent (FTA)**

  A workstation capable of resolving local dependencies and launching its jobs in the absence of a domain manager.

### 1.3 Topology

A key to choosing how to set up Tivoli Workload Scheduler domains for an enterprise is the concept of localized processing.

The idea is to separate or localize the enterprises’ scheduling needs based on a common set of characteristics.

Common characteristics are things such as geographical locations, business functions, and application groupings.
Grouping related processing can limit the amount of interdependency information that needs to be communicated between domains.

The benefits of localizing processing in domains are:

- Decreased network traffic. Keeping processing localized to domains eliminates the need for frequent interdomain communications.

- Provides a convenient way to tighten security and simplify administration. Security and administration can be defined at, and limited to, the domain level. Instead of network-wide or workstation-specific administration, it is possible to have domain administration.

- Network and workstation fault tolerance can be optimized. In a multiple domain Tivoli Workload Scheduler network, it is possible to define backups for each domain manager, so that problems in one domain do not disrupt operations in other domains.

1.4 Tivoli Workload Scheduler scheduling objects

Scheduling with Tivoli Workload Scheduler includes the capability to do the following:

- Schedule jobs across a network.

- Group jobs into job streams according to, for example, function or application.

- Set limits on the number of jobs that can run concurrently.

- Create job streams based on day of the week, on specified dates and times, or by customized calendars.

- Ensure correct processing order by identifying dependencies such as successful completion of previous jobs, availability of resources, or existence of required files.

- Set automatic recovery procedures for unsuccessful jobs.

- Forward incomplete jobs to the next production day.

Starting from version 8.3, the Tivoli Workload Scheduler scheduling objects are stored in a relational database.

This results in a significant improvement, in comparison with previous versions, of how objects are defined and managed in the database.

Each object can now be managed independently without having to use lists of scheduling objects such as calendars, parameters, prompts, and resources.

The command syntax used to define and manage these objects has also become direct and powerful.

Tivoli Workload Scheduler administrators and operators work with these objects for their scheduling activity:
- **Workstation.** Also referred to as CPU. Usually an individual computer on which jobs and job streams are run. Workstations are defined in the Tivoli Workload Scheduler database as unique objects. A workstation definition is required for every computer that runs jobs or job streams in the Tivoli Workload Scheduler network.

- **Workstation class.** A group of workstations. Any number of workstations can be placed in a class. Job and job streams can be assigned to execute on a workstation class. This makes replication of a job or job stream across many workstations easy.

- **Job.** A script or command, run on the user’s behalf, run and controlled by Tivoli Workload Scheduler.

- **Job stream.** A list of jobs that run as a unit (such as a weekly backup application), along with run cycles, times, priorities, and other dependencies that determine the exact order in which the jobs run.

- **Calendar.** A list of scheduling dates. Each calendar can be assigned to multiple job streams. Assigning a calendar to a job stream causes that job stream to run on the dates specified in the calendar. A calendar can be used as an inclusive or as an exclusive run cycle.

- **Run cycle.** A cycle that specifies the days that a job stream is scheduled to run. Run cycles are defined as part of job streams and can include calendars that were previously defined. There are three types of run cycles: a Simple run cycle, a Weekly run cycle, or a Calendar run cycle (commonly called a calendar). Each type of run cycle can be inclusive or exclusive, that is, each run cycle can define the days when a job stream is included in the production cycle, or when the job stream is excluded from the production cycle.

- **Prompt.** An object that can be used as a dependency for jobs and job streams. A prompt must be answered affirmatively for the dependent job or job stream to launch. There are two types of prompts: predefined and ad hoc. An ad hoc prompt is defined within the properties of a job or job stream and is unique to that job or job stream. A predefined prompt is defined in the Tivoli Workload Scheduler database and can be used by any job or job stream.

- **Resource.** An object representing either physical or logical resources on system. After being defined in the Tivoli Workload Scheduler database, resources can be used as dependencies for jobs and job streams. For example, it is possible to define a resource named tapes with a unit value of two, and then, define jobs that require two available tape drives as a dependency. Jobs with this dependency cannot run concurrently because each time a job is run the tapes resource is in use.

- **Variable and variable table.** A variable can be used to substitute values in scheduling objects contained in jobs and job streams, that is, in JCL, log on, prompts dependencies, file dependencies, and recovery prompts. The values are replaced in the job scripts at run time.
Variables are global (that is, they can be used on any agent of the domain) and are defined in the database in groups called variable tables.

- **Parameter.** A parameter can be used to substitute values in jobs and job streams just like global variables. The difference is that a parameter is defined on the specific workstation where the related job is to run and has no global effect, but affects only that specific workstation. Parameters cannot be used when scripting extended agent jobs.

- **User.** On Windows workstations, the user name specified in the Logon field of a job definition must have a matching user definition. The definitions provide the user passwords required by Tivoli Workload Scheduler to launch jobs.

- **Event rule.** A scheduling event rule defines a set of actions that are to when specific event conditions occur. The definition of an event rule correlates events and triggers actions. When an event rule is defined, it specifies one or more events, a correlation rule, and one or more actions that are triggered by those events. Moreover, it is possible to specify validity dates, a daily time interval of activity, and a common time zone for all the time restrictions that are set.

It is possible to control how jobs and job streams are processed with the following attributes:

**Dependencies** - Conditions that must be satisfied before a job or job stream can run. It is possible to set the following types of dependencies:

- A predecessor job or job stream must have completed successfully.
- One or more specific resources must be available.
- Access to specific files must be granted.
- An affirmative response to a prompt.

**Time constraints** - Conditions based on time, such as:

- The time at which a job or job stream should start.
- The time after which a job or job stream cannot start.
- The repetition rate at which a job or job stream is to be run within a specified time slot.

**Job priority** - A priority system according to which jobs and job streams are queued for execution.

**Job fence** - A filter defined for workstations. Only jobs and job streams whose priority exceeds the job fence value can run on a workstation.

**Limit** - Sets a limit to the number of jobs that can be launched concurrently on a workstation.
1.5 Graphical view of Tivoli Workload Scheduler components involved in the capacity planning test activities

This section provides a graphical illustration of the subset of the components involved in the capacity planning test activities, of those numbered in the previous sections.

As can be seen from the previous figure, the components involved in the capacity planning activities are, mainly:

- Tivoli Workload Scheduler Master Domain Manager (engine), acting as the engine that orchestrates and monitors the execution, by the agents, of the jobs making up the workload
- Fault-tolerant agents, acting as the components responsible for running the jobs in the plan, and notifying the job execution results to the engine itself.

2 Goal

The purpose of the Tivoli Workload Scheduler V8.5.1 engine Capacity Planning test activity is to provide information and answers to a customer that needs to identify and size a suitable environment to successfully manage their required workload. By reading this document, a customer should be able to decide, configure, and set up an environment, in terms of hardware servers provided with appropriate features, and network infrastructure,
so that the workload, in terms of jobs to be run, can be successfully managed by the Tivoli Workload Scheduler V8.5.1 engine.

To help the customer to decide the correct environment configuration, for a given workload, the capacity plan test activity is composed of two orthogonal parameters: Tivoli Workload Scheduler engine server machine configuration and workload size.

The Tivoli Workload Scheduler engine was installed on a server machine provided with the possibility of changing the hardware configuration. The Tivoli Workload Scheduler engine server machine was built required on a P6 Server machine with the capability of hosting virtual machines with the required features, and the possibility of changing the configuration of a virtual machine, modifying the number of core processors and the RAM available.

Although the server on which Tivoli Workload Scheduler engine was installed is located on a virtual machine, the virtualization environment means that resources for the server are dedicated and not shared, so that performance cannot be affected by any other virtual machine hosted by the same P6 virtualization infrastructure.

In this way, it was possible to run a given workload on 5 distinct Tivoli Workload Scheduler engine server hardware configurations, without the need for switching to a distinct physical machine provided with different hardware features.

The other orthogonal parameter considered in this test activity is the workload; with the purpose of providing the customer with the most comprehensive set of information, test activity was run on 3 distinct workloads:

- Light workload
- Medium workload
- Heavy workload

Each workload differs from the previous in the number of scheduling objects making up the workload itself. Light workload has a small set of scheduling objects; medium workload is provided with an intermediate set of scheduling objects; heavy workload has the biggest number of scheduling objects in this test activity. Further details about the workloads are available in the corresponding Workload section.

Then, considering the fact that Tivoli Workload Scheduler engine server configurations and workloads are orthogonal, it was possible to provide a set of 15 distinct test cases, matching each of the 5 server configurations with each of the 3 workloads.

<table>
<thead>
<tr>
<th>Configuration/Workload</th>
<th>Light Workload (WI)</th>
<th>Medium Workload (Wm)</th>
<th>Heavy Workload (Wh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config1 (C1)</td>
<td>C1-Wl</td>
<td>C1-Wm</td>
<td>C1-Wh</td>
</tr>
<tr>
<td>Config2 (C2)</td>
<td>C2-Wl</td>
<td>C2-Wm</td>
<td>C2-Wh</td>
</tr>
<tr>
<td>Config3 (C3)</td>
<td>C3-Wl</td>
<td>C3-Wm</td>
<td>C3-Wh</td>
</tr>
<tr>
<td>Config4 (C4)</td>
<td>C4-Wl</td>
<td>C4-Wm</td>
<td>C4-Wh</td>
</tr>
</tbody>
</table>
Each test case was coded into a short string in the format as C<n>-W<m>, where <n> refers to the particular configuration that the Tivoli Workload Scheduler engine server is set on, and <m> is the type of workload, coming from the set {Light, Medium, Heavy}.

For example, the test case “C4-Wm” refers to the execution of the Medium Workload against the Tivoli Workload Scheduler engine on the server machine configured with the Config3.

### Approach

#### 3.1 Installation and setup

The first preliminary phase of the Tivoli Workload Scheduler V8.5.1 engine Capacity Planning test activity was the installation and the setup of the entire environment.

To obtain a full working environment for the purpose of the capacity planning activities, the following components were installed:

- Tivoli Workload Scheduler V8.5.1 engine (Global Availability code level), installed following the normal graphical installation, in the default path, on a dedicated server machine, starting from a scratch server clean configuration.

- DB2 9.5.0.1 Database (DB2), installed following the normal graphical installation, in the default path, on a dedicated distinct server machine, starting from a scratch server clean configuration.

- Tivoli Workload Scheduler fault-tolerant agents, 32 instances installed on 8 distinct servers, with 4 instances shared per single physical server machine, using silent installation, with response files automatically generated by using “ad hoc” scripting to automate and speed up the whole installation process.

All the components were installed in clean environments, that is, on scratch server machines dedicated to host only the components described above, so that no other significant application (in terms of resource consumption) runs on the same machines during the test process.

Where possible, snapshots of virtualized machines and databases were taken after the components were successfully installed, to make it possible to easily revert to a safe initial clean configuration at any new test start, restoring the exact initial conditions for each test run during the capacity plan activity.
3.2 Test initial conditions

To perform test activities in the same conditions for all runs, so that the only difference between the current run and the previous one is an intentionally modified feature (for example, configuration or workload), the following setup protocol was followed, where applicable:

- Stop the current applications
- Clean all the traces, logs, and written data on the application server machine and store them if needed
- Check that machine health status is OK (free memory, RAM, CPU, disk space, and so on)
- Reboot the system machine, if needed
- Restart the application to obtain a clean scratch environment

3.3 Database information and tuning settings

All the tests were run using a dedicated database server machine. The database application was IBM DB2 Version 9.5.0.1.

Database DB2 was configured with the following tuning modifications, to improve and optimize performances when used with Tivoli Workload Scheduler V8.5.1.

For Tivoli Workload Scheduler DB:

Log file size (4KB) (LOGFILSZ) = 10000
Number of primary log files (LOGPRIMARY) = 80
Number of secondary log files (LOGSECOND) = 40

For embedded WebSphere Application Server:

<connectionPool xmi:id="ConnectionPool_1174319908875" connectionTimeout="180" maxConnections="50" minConnections="1"
reapTime="180" unusedTimeout="1800" agedTimeout="0"
purgePolicy="EntirePool" numberOfSharedPoolPartitions="0"
numberOfUnsharedPoolPartitions="0" numberOfFreePoolPartitions="0"
freePoolDistributionTableSize="0" surgeThreshold="-1"
surgeCreationInterval="0" testConnection="true"
testConnectionInterval="15" stuckTimerTime="0" stuckTime="0"
stuckThreshold="0"/>

The database component was not changed, in terms of configuration or any other modifiable parameter, during the entire test activity for capacity plan; some database server machine reboots were sometimes made, to ensure that the server was always healthy with no configuration/hardware problems and issues.
The only differences in the database, between one particular test case and another one, could only were the particular workload, in terms of scheduling objects present in the database itself.

4 Environment

The Tivoli Workload Scheduler V8.5.1 engine Capacity Plan environment, with the purpose of collecting data and information about engine server machine performance and resource consumption under a specific workload test, is composed of two main components: Tivoli Workload Scheduler V8.5.1 engine with database DB2 and Tivoli Workload Scheduler V8.5.1 fault-tolerant agents.

4.1 Tivoli Workload Scheduler V8.5.1 engine

The Tivoli Workload Scheduler V8.5.1 engine is the Capacity Plan target component, on which we want to know resource consumption information during a certain workload test.

The engine is implemented by a unique server, hosted by an AIX 6.1.4.0 System (64-bit architecture), virtualized over a P6 server environment.

Tivoli Workload Scheduler V8.5.1 engine hardware resources are dedicated, so that they cannot be modified, influenced, or changed by any other system hosted by the same P6 server environment.

The virtualization allows the engine server configuration to be dynamically changed, in terms of amount of core and available RAM, leading to a total number of 5 different engine configurations, detailed in the table below:

<table>
<thead>
<tr>
<th>Config</th>
<th>CPU</th>
<th>RAM</th>
<th>SWAP</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>1 x 2 PowerPC_POWER6 4204 MHz</td>
<td>2 GB</td>
<td>4 GB</td>
<td>150 GB</td>
</tr>
<tr>
<td>C2</td>
<td>2 x 2 PowerPC_POWER6 4204 MHz</td>
<td>2 GB</td>
<td>4 GB</td>
<td>150 GB</td>
</tr>
<tr>
<td>C3</td>
<td>2 x 2 PowerPC_POWER6 4204 MHz</td>
<td>4 GB</td>
<td>4 GB</td>
<td>150 GB</td>
</tr>
<tr>
<td>C4</td>
<td>4 x 2 PowerPC_POWER6 4204 MHz</td>
<td>2 GB</td>
<td>4 GB</td>
<td>150 GB</td>
</tr>
<tr>
<td>C5</td>
<td>4 x 2 PowerPC_POWER6 4204 MHz</td>
<td>4 GB</td>
<td>4 GB</td>
<td>150 GB</td>
</tr>
</tbody>
</table>

Table 2 - Tivoli Workload Scheduler 8.5.1 engine configuration descriptions

Each configuration is provided with a fixed value of 4 GB of swap paging space, and of a hard disk of 150 GB.

The variable features, configuration by configuration, are the number of processor cores (1, 2 and 4) and the amount of RAM (2 GB or 4 GB).

Processor type is always the same, that is a Power6 with a clock frequency of 4,204 MHz; the variable is the number of cores available, that was changed in the set \{1,2,4\}.
Configurations are ordered by number of processor cores and, then, by amount of RAM. This gives a sort of power order for the configurations, so that C1 is the weakest configuration, while C5 is the most powerful configuration.

Configuration C3 and C4 can be compared to be, from a pre-test activity point of view, at a similar power level, because the first has 2 cores and 4 GB of RAM, while the latter has 4 cores and 2 GB of RAM.

The engine code version installed on this machine is the 8.5.1 GA (Global Availability); engine has also the embedded WebSphere Application Server (embedded WebSphere Application Server) version 6.1.0.23 (FP23) installed.

Both the Tivoli Workload Scheduler V8.5.1 engine and embedded WebSphere Application Server 6.1.0.23 are components built over a 32-bit Java Virtual Machine with the following features: IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 AIX).

4.1.1 Tivoli Workload Scheduler engine embedded WebSphere Application Server heap size configuration and tuning

An important tuning is related to the heap size of the embedded WebSphere Application Server Java process. This parameter is used to configure the amount of primary RAM memory that the embedded WebSphere Application Server can use to allocate the objects that it needs to manage during normal working activities.

It is possible to set the initial size of the heap memory that is the amount of memory that the embedded WebSphere Application Server Java process takes for itself, and the maximum size of heap memory, that is the maximum amount of memory the embedded WebSphere Application Server Java process can get, whenever needed.

It is important to remember that the amount of heap size, both for initial and maximum values, is upper-bounded by 2 main factors:

- Server machine physical RAM available
- 32-bit address space limit of JVM

This means that it is not possible to set any value for the heap size, but that there is an upper bound that cannot be exceeded (otherwise, embedded WebSphere Application Server would not start up).

The choice for initial and maximum heap size was made by choosing a value that is compliant with both the constraints described above.

The upper bound for 32-bit based JVM architectures is about 1.5 GB.

On the other side, it is a well-known performance best practice not to reserve the entire system RAM available to the Java embedded WebSphere Application Server process, but only a reasonable percentage, so that other system processes, necessary for a healthy system status, can run correctly with no resources availability issues.

Then, the formula used to compute the initial and maximum heap size for the capacity planning activity, was to use:
- Max heap size: minimum value between 50% of total available RAM and the 32-bit addressing limit of 1.5 GB

- Min heap size: the maximum heap size value decreased by 0.5 GB (512 MB).

This formula led to the following heap values, for all the 5 configurations, summarized in the following table:

<table>
<thead>
<tr>
<th>Config</th>
<th>RAM</th>
<th>initial heap size</th>
<th>maximum heap size</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>2.0 GB</td>
<td>0.5 GB</td>
<td>1.0 GB</td>
</tr>
<tr>
<td>C2</td>
<td>2.0 GB</td>
<td>0.5 GB</td>
<td>1.0 GB</td>
</tr>
<tr>
<td>C3</td>
<td>4.0 GB</td>
<td>1.0 GB</td>
<td>1.5 GB</td>
</tr>
<tr>
<td>C4</td>
<td>2.0 GB</td>
<td>0.5 GB</td>
<td>1.0 GB</td>
</tr>
<tr>
<td>C5</td>
<td>4.0 GB</td>
<td>1.0 GB</td>
<td>1.5 GB</td>
</tr>
</tbody>
</table>

*Table 3 - Heap size initial and maximum values for each Tivoli Workload Scheduler engine server configuration*

### 4.2 DB2 Database

The Tivoli Workload Scheduler V8.5.1 engine needs a database in which to store the Tivoli Workload Scheduler scheduling object definitions.

The database used for the Capacity Plan test is IBM DB2 V 9.5.0.1 for AIX.

The database is hosted by a distinct AIX Server machine, also in this case hosted by the P6 virtualization environment, with the following features:

<table>
<thead>
<tr>
<th>Machine</th>
<th>CPU</th>
<th>RAM</th>
<th>SWAP</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>3 x 2 PowerPC_POWER6 4204 MHz</td>
<td>4 GB</td>
<td>4 GB</td>
<td>150 GB</td>
</tr>
</tbody>
</table>

*Table 4 - Database server machine details*

Then, the database server machine is provided with 3 process cores Power6 with clock frequency of 4,204 MHz, 4 GB of RAM and a swap space of 4 GB; finally, server is provided with a 150 GB hard drive.
4.3 Tivoli Workload Scheduler V8.5.1 fault-tolerant agents

The Tivoli Workload Scheduler V8.5.1 fault-tolerant agent component is responsible for running the specific test workload (as jobs) and to provide notifications to the engine about the execution status, at any change.

The purpose of this component is to create and run the actual workload that the Tivoli Workload Scheduler engine needs to manage during a test.

Tivoli Workload Scheduler V8.5.1 fault-tolerant agents are spread over 8 virtualized server machines Linux RedHat 5.3 (64-bit architecture), hosted by VMware virtualization center.

Tivoli Workload Scheduler V8.5.1 fault-tolerant agent hardware resources are dedicated, so that they cannot modified, influenced, or changed by any other system hosted on the same virtualization center.

To provide the Tivoli Workload Scheduler engine with a balanced workload, all the server machines have the same hardware configuration, as shown in the following table:

<table>
<thead>
<tr>
<th>OS</th>
<th>CPU</th>
<th>RAM</th>
<th>SWAP</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHEL 5u3</td>
<td>4 x Intel® Xeon® CPU X7460 @2.66GHz</td>
<td>10 GB</td>
<td>2 GB</td>
<td>200 GB</td>
</tr>
</tbody>
</table>

*Table 5 - Tivoli Workload Scheduler 8.5.1 fault-tolerant agent Server machine description*

The main hardware features, of each server, are: processor with 4 cores (Interl Xeon X7460 with clock frequence of 2.66 Ghz), 10 GB of RAM, 2 GB of swap paging space, and hard disk provided with 200 GB.
Each fault-tolerant agent machine hosts 4 distinct instances of Tivoli Workload Scheduler V8.5.1 fault-tolerant agents, numbered from 0 to 3 (0, 1, 2, and 3).

The generic Tivoli Workload Scheduler V8.5.1 fault-tolerant agent instance is named as fault-tolerant agent_xxx_<n>, where xxx is a value corresponding to the name of the hosting machine (023, 033, 038, 050, 056, 058, 059, 120), and <n> is a number from {0, 1, 2, 3} referring to a specific fault-tolerant agent instance hosted on that particular machine.

For example, fault-tolerant agent_050_3 is the fourth Tivoli Workload Scheduler V8.5.1 fault-tolerant agent instance hosted by the fault-tolerant agent_050 machine.

### 4.3.1 Fault-tolerant agent full status configuration

Fault-tolerant agents can be configured by setting several parameters to modify some features.

One important setting is related to “full status”, that is the capability of an agent to exchange status messages with peers (other agents in the same domain), to be able to know information about job execution and results from the other peers.

With the full status feature enabled, a fault-tolerant agent not only communicates with the Tivoli Workload Scheduler engine, but also with all the other agents involved in the plan execution. Enable this feature if a fault-tolerant agent could be potentially a Backup Domain Manager.

During the capacity planning test activities, it was decided to enable the full status feature, because this feature has a non negligible impact on performance, because it increases the number of messages exchanged in the network infrastructure, because the agent communicates with all the peers, in addiction to the engine.

In this way, with the full status enabled, it was possible to run a worst-case analysis, because of the overhead on the network, on the agents, and on the engine due to the fact that fault-tolerant agents exchange messages among each other.

A performance upper-bound was found so that, in an actual production environment, where the full status is supposed to be switched off (unless an agent is a candidate for being a Backup Domain Manager), the performance will be better than the provided case.

It is interesting to note that, during the Execution phase, with the Heavy Workload (the largest one), it was impossible to complete the whole plan execution with the full status enabled, because of the fact that the agents and engine message queues were saturated by the excessive number of messages exchanged. This is why under these conditions it was necessary to disable the full status.

Moreover this is the reason why, in the results test, the Heavy Workload resource usage is a little lighter, different, and unexpected considering the trend that could be traced in conjunction with the other 2 workload-obtained resource usage values.

In other words, with the full status disabled on the third configuration, some test results can be better, compared with the expected ones deduced from the trend of the other configurations (having the full status enabled).

The following table summarizes the full status configuration for any workload provided.
### 4.3.2 Mailman servers and message queues

Mailman servers allocate separate processes dedicated to the communication with other workstations. The main mailman is dedicated to the transfer and network hub activities. For these test activities, 4 Mailman servers were planned to manage all 32 fault-tolerant agents, that is one mailman server is connected to 8 fault-tolerant agents.

Planning space for queues is another critical factor. Because flows are greater at the master domain manager and at any full status fault-tolerant agents in the master domain, the queue size were changed from 10 MB to 50MB.

### 4.4 Network

The Tivoli Workload Scheduler V8.5.1 engine communicates with the DB server machine and fault-tolerant agent machines over a 1 Gbit Ethernet network and adapters.

This means that, under normal working conditions of the network infrastructure, no bottleneck is expected from the links connecting the distinct components involved in this test activities, as the bandwidth is enough to allow strong traffic and data exchanges.

The following schema summarizes the Tivoli Workload Scheduler V8.5.1 engine Capacity Plan:

<table>
<thead>
<tr>
<th>Workload</th>
<th>Fault-tolerant agent full status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>Enabled</td>
</tr>
<tr>
<td>Medium</td>
<td>Enabled</td>
</tr>
<tr>
<td>Heavy</td>
<td>Disabled</td>
</tr>
</tbody>
</table>

*Table 3 - Workload configuration and full status related setting*
Scenarios

The left side shows the P6 virtualized environment, which hosts the configuration-changeable server that contains the Tivoli Workload Scheduler V8.5.1 engine, and the database server, that contains the IBM DB2.

The right side shows the VMware virtualized environment, which hosts the 8 fault-tolerant agent server machines, all provided with the same hardware configuration; each fault-tolerant agent server machine hosts 4 distinct Tivoli Workload Scheduler fault-tolerant agent instances.

fault-tolerant agents communicate with the Tivoli Workload Scheduler engine through a Gigabit Ethernet network; the Tivoli Workload Scheduler engine communicates with the DB2 database over a Gigabit Ethernet network.

5 Scenarios

Tivoli Workload Scheduler V8.5.1 engine Capacity plan testing consists of a sequence of scenarios to be run and to collect data results about the Tivoli Workload Scheduler engine server hardware resource usage.

Scenarios consist of the submission of a certain workload on the engine, and to monitor how the server itself, under a certain configuration, is affected by the particular workload.
To drill down, for a given workload, and a given Tivoli Workload Scheduler engine server configuration, the Capacity plan consists of 4 distinct test scenarios:

1. Plan generation (**JnextPlan**) 
2. Plan execution and static scheduling (**Execution**) 
3. Plan extension and further tests (**Extension**) 
4. Dynamic job submission (**Submit**) 

All the scenarios are run after a preliminary phase that consists of setting up the engine and populating its database with all the necessary scheduling objects.

The first three tests were run in a serial way, for each workload and each configuration, after the initial setup phase in which the scheduling objects were created through script automation and inserted in the DB for population, via composer interface.

The fourth test (dynamic job submission), was run separately, and later, after starting again on a restored full clean environment (as in a just-installed product status), because the scheduling objects used for database definition populations are different, as designed in the test design phase.

All the scenarios are repeated for each workload level and each Tivoli Workload Scheduler engine server configuration, for a total number of 15 cases (5 server configurations per 3 workloads).

### 5.1 Engine setup and database population

To make the Tivoli Workload Scheduler engine able to schedule and monitor jobs execution on the fault-tolerant agent machines, the scheduling object definitions must be loaded on the engine database.

This is the phase where the workload (the scheduling objects) is actually provided to the engine.

In this way, the Tivoli Workload Scheduler engine can know:

- Which agents can execute jobs, with all their features
- Which jobs need to be executed, and all their features
- Which job streams contain the jobs to execute, and all their features
- Which prompt and machine resources are available, and all the details about them.

This is the engine database population; when the database was successfully populated, the engine knows the entire scheduling environment and, if the fault-tolerant agents are ready, it can start scheduling jobs.
Composer command line takes as input scheduling object descriptions, written in a particular syntax, validates them and, if no error is found, loads the input data into the engine database.

Composer can be used as an interactive environment to load, modify, or delete data against the engine database, or it can be used directly by passing scheduling object definitions as input files.

In the Tivoli Workload Scheduler V8.5.1 engine Capacity Plan, working with workloads containing a very large number of scheduling objects (jobs, job streams, dependencies, resources, and so on), it was necessary to work with composer not in the interactive way, but by passing it directly files containing the scheduling objects written in the correct syntax.

To generate the composer scheduling object input definitions, some automation scripts were used which by getting a few parameters as input, for example number of jobs per job streams, number of job streams per fault-tolerant agent and fault-tolerant agent instances list, are able to create the entire composer input files containing the scheduling object definitions; in other words, automation scripts allow the generation in a simple way of the Capacity Plan workloads as composer input files.

![Figure 5 - Process to automate the input generation for scheduling object creation in TWS database, to generate the workload](image)

5.2 Plan generation (JnextPlan)

The first part of the actual Capacity Plan test scenario is the plan generation performed using the Tivoli Workload Scheduler JnextPlan command, which produces, as output, a file called Symphony, containing the production plan.

This file is then distributed to all the agents responsible for running jobs.

The plan generation is an activity completely done by the Tivoli Workload Scheduler engine, and it will require time and CPU resources depending on the size of the definitions in the engine database.

Some monitoring tools and logs were used to track the time required, and the engine server machine hardware resources used by the Tivoli Workload Scheduler to prepare the plan with the JnextPlan command:
Scenarios

- nmon tool: to monitor the Tivoli Workload Scheduler engine Server system resource usage
- ad hoc tool: to monitor the memory usage by the Tivoli Workload Scheduler embedded WebSphere Application Server Java process
- Modified version of JnextPlan able to write to a file the duration (completion time – start time) of the plan preparation process
- embedded WebSphere Application Server logs (SystemOut, SystemErr)
- Tivoli Workload Scheduler V8.5.1 engine logs and traces (Tivoli Workload Scheduler stdlist folder content with logs and traces)

It is important to assert that, before running the JnextPlan command, the initial condition is that the plan does not exist (if necessary, using the command “ResetPlan –scratch” to delete the current production plan).

By changing the workload and/or the configuration in this way, what is monitored is always the same that is, the plan is prepared from a scratch configuration.

After the production plan was generated, the recorded engine server hardware resource usage and logs were collected and stored for a subsequent result analysis.

A further drilldown in terms of timings and durations of the plan generation are described in more detail the results section, that is:

- MakePlan phase:
  - Long Term Plan (LTP) creation phase
  - Current Plan (CP) creation phase

Finally, it is useful to note that the JnextPlan command was also modified to skip the statistics creation part on old plans, to speed up the procedure and to focus only on the current plan creation subtask.

5.3 Plan execution and static scheduling (Execution)

When the Plan was generated, the environment is ready for job execution by the fault-tolerant agents and for the planning and scheduling by the Tivoli Workload Scheduler engine.

This scheduling activity is referred to as static because no additional jobs are submitted to the engine during the plan execution, but only jobs in the production plan are actually run.

To trigger the fault-tolerant agents to start running jobs in the plan, the fault-tolerant agents limit must be increased, that is, the maximum number of jobs that a fault-tolerant agent instance can run simultaneously on the machine hosting the agent.
The limit was set to 512; a higher limit value might overload the fault-tolerant agent machine is, for example, in a typical customer use case, a machine that hosts an fault-tolerant agent is not completely dedicated to it, but can do other tasks at the same time.

As soon as the limit was increased up to 512 for each fault-tolerant agents (and the engine acting as agent), the monitoring tools and logs were activated on the engine server machine.

These monitors were switched off only when the plan execution was completed.

To avoid a possible bottleneck, fault-tolerant agents were constantly monitored, using some virtualization resources monitoring tools.

This ensured that the test results were not affected by bottlenecks due to fault-tolerant agents resource saturation, which could cause incorrect engine server usage data.

To manage the duration of the plan execution and be able to handle the results, the plan was set up with time restrictions defined at job streams level (the “AT hhmm” dependency), which allows the start time of each job stream to be specified, within a particular time interval, between 12.00 pm and 5.00 pm (5 hour time interval).

Further details about the job streams time restrictions can be found in the Workload section.

The actual job execution by the agents was started 1 hour after the “AT hhmm” (where “hh” indicates an hour of the day expressed in two digits, in the interval 0-23, and “mm” indicates a minute of the hour, expressed in two digits, in the interval 0-59) condition initial time interval set in the workload, that is at 1.00 pm (this behavior was deliberately chosen to create a stress condition to the engine server machine and analyze the resource usage during this phase).

The stress condition comes from the fact that if the time specified in a job streams’ “AT hhmm” condition is exceeded when changing the fault-tolerant agents’ limit from 0 to 512, the agent could possibly run the jobs inside that job streams immediately (or as soon as possible, depending on other existing constraints and dependencies), because the designed execution time has already passed.

This highlights the fact that all the job streams with an “AT hhmm” time condition in the first hour of execution time (from 12.00 am to 1.00 pm) must be executed by the fault-tolerant agents as soon as possible, because their execution time has already passed.

The “AT hhmm” condition in the job streams, as described in the following Workload section, is set in a random way, picking a random time in the interval from 12.00 am to 1.00 pm, with the same probability (then, with a uniform distribution of values).

So, it is expected that the number of job streams having an “AT hhmm” time condition between 12.00 am and 1.00 pm is about 20% over the total number, because this interval is 1 hour over the 5 hours of the total execution time).

At 2.30 pm, jobs waiting for a prompt reply (see Workload section, were released by answering yes to the prompt.

In this way, those jobs can be successfully executed and allow the execution of all the jobs depending on them through a “follows dependency”. 
At that point, all the jobs can be executed in compliance with the job streams “AT hh:mm” time conditions and all the existing dependencies constraints.

The Plan execution phase is considered completed as soon as all the jobs and, as a consequence, the job streams go into “Success” status.

To monitor the plan execution progress, Tivoli Workload Scheduler conman command line was used to query the Tivoli Workload Scheduler engine about the job and job stream statuses.

In detail, conman was used to get the number of jobs and job streams in successful status during the Plan execution time, and to check the effective completion condition.

After the plan execution has completed, recorded engine server hardware resource usage were collected and stored for a subsequent result analysis, provided in the related section.

The following picture summarizes the Tivoli Workload Scheduler V8.5.1 engine Capacity Plan test scenarios JnextPlan and Execution phases.
5.4 Plan extension and further tests (Extension)

With the purpose of providing a more precise and detailed set of results, the plan generation activity was repeated for a number of trials, to provide results containing minimum errors in the measurements, might have been possible to avoid issues coming from a single test execution (obtaining a results that can be affected and modified by a particular circumstance or external unpredictable factor).
In this way, the plan generation task was repeated, for each Tivoli Workload Scheduler engine server configuration, and for each workload level, 5 times.

For each plan generation task, the same pre-test conditions were restored, to provide effectively comparable results.

In more details, before running the JnextPlan command, this step sequence was performed:

1. Stop any active monitoring tools such as nmon and collecting old data
2. Clean logs if necessary
3. Reset current plan using the "ResetPlan –scratch" command
4. Activate monitoring tools such as nmon

The JnextPlan command has now been run, with the previously described modification to print the timestamp of the command start and completion time in a file.

Leveraging on messages in the embedded WebSphere Application Server SystemOut logs, it was also possible to discover the start and completion times of MakePlan and its sub activities the JnextPlan task. After the JnextPlan plan generation from scratch was repeated 5 times, a new task was run. This new task extends the current production plan to a new one.

The JnextPlan extension was repeated 3 times, starting from the produced plan at the end of the fifth iteration of the first phase of this test activity.

As can be seen in detail in the results section, this task did not complete successfully every time it ran (on each configuration and each workload), because of the occurrence of some Java OutOfMemory exceptions. The following images show, in a graphical way the process of this test activity.
Scenarios

Figure 7 - JnextPlan further tests and extension details

Figure 8 - JnextPlan extension part details
5.5 Dynamic job submission (Submit)

The dynamic job submission test activity has the purpose of verifying the behavior of the Tivoli Workload Scheduler engine when new jobs are submitted, in addition to the ones already present in the current production plan.

In more details, this test checks, for a given time interval of 3 hours, the job submission rate, that is, how many jobs the engine can manage in the analyzed time interval.

The interval of 3 hours was chosen to try to understand if the job submission rate is constant or if it changes as time goes by for any reason. A reasonable interval of time, long enough to detect such a similar trend, was chosen for testing.

It is useful to recall the fact that this test, differently from the previous 3 test activities, is based on a different and ad hoc set of scheduling objects, having different topologies and constraints, as designed in the test preparation.

This required an initial setup phase to restore the environment to a clean “post-installation” status, for all the components.

The initial setup phase is composed of this step sequence:

1. Restore to a fully clean environment: Tivoli Workload Scheduler DB for scheduling objects is empty, Tivoli Workload Scheduler engine is as at “just installed time” (empty Tivoli Workload Scheduler traces and logs, no embedded WebSphere Application Server SystemOut and SystemErr traces, empty queues), server is healthy (CPU is OK, RAM is free, there is enough space on hard disk drive); same conditions for each fault-tolerant agent instance.

2. Start up the whole environment (Tivoli Workload Scheduler engine and fault-tolerant agents).

3. Create with automation scripts appropriate scheduling objects for this dynamic job submission scenario (details in the Workload section), and add them by using composer for database population. In particular, scheduling objects can be split into those that comprise the plan and in those that are the actual jobs to be dynamically submitted.

4. Generate the plan with the JnextPlan command.

The environment is now ready for the test execution. The test was repeated for each of the 5 configurations used in this test, for the Tivoli Workload Scheduler engine server.

Each test was run in 2 different ways:

- With fault-tolerant agent not executing the jobs in plan (limit set to 0) – only submission
- With fault-tolerant agent executing the jobs in plan (limit set to 512) – plan execution
This was done to check if the actual plan execution affects the rate of jobs submitted and successfully managed by the Tivoli Workload Scheduler engine.

The test consists of automation scripts that submit in a consecutive way, inside a loop that is interrupted after 3 hours, 10 job streams, each of them containing 10 jobs; further details can be found in the Workload section.

For each test, monitoring tools were activated to analyze the resource usage. At the end of each test, this data were collected:

- nmon system performance monitoring data plus ad hoc tool to monitor Tivoli Workload Scheduler WAS Java process memory usage
- embedded WebSphere Application Server logs (SystemOut and SystemErr)
- Tivoli Workload Scheduler engine logs (stdlist, logs and traces)

The following image summarizes the entire dynamic job submission test activity, in a graphical way.
6 Workload

6.1 Introduction

The Tivoli Workload Scheduler V8.5.1 engine Capacity Plan test consists of collecting information about the hardware resource usage of the server machine that hosts the Tivoli Workload Scheduler V8.5.1 engine, while a certain workload is running.

This is done to provide configuration details to help a customer to set up an appropriate environment able to manage the required workload in terms of jobs, job streams, and dependencies.

Dependencies can of course be defined in very complex modes, through job and job streams, but considering the performance and scalability purposes of this document, only simple dependencies and constraints cases are described.

6.2 Workloads for plan preparation, plan execution, and JnextPlan further test and extensions

This section and its subsections describe the workload used for the plan generation (JnextPlan), the plan execution, and the JnextPlan further tests.

Tivoli Workload Scheduler V8.5.1 engine Capacity Plan Workload can be briefly summarized by highlighting 3 parameters:

- Number of jobs
- Number of job streams
- Number of Dependencies

The Capacity Plan for Tivoli Workload Scheduler V8.5.1 engine was run using three different workloads: Workload Light (WL), Workload Medium (WM), and Workload Heavy (WH).

The following table provides details about the 3 workload parameters:

<table>
<thead>
<tr>
<th>Workload name</th>
<th>Jobs</th>
<th>Job streams</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload Light (WL)</td>
<td>100000</td>
<td>5000</td>
<td>200000</td>
</tr>
<tr>
<td>Workload Medium (WM)</td>
<td>200000</td>
<td>10000</td>
<td>250000</td>
</tr>
<tr>
<td>Workload Heavy (WH)</td>
<td>300000</td>
<td>15000</td>
<td>300000</td>
</tr>
</tbody>
</table>

*Table 4 - Number of jobs, job streams, and dependencies per workload*

For all three workloads, the total number of scheduling objects was divided always with these percentages:

- 10% of the workload on the Tivoli Workload Scheduler Master acting as agent
90% of the workload similarly distributed on the 32 fault-tolerant agents instances

In this way, Tivoli Workload Scheduler Master runs 10% of the total current workload, while each fault-tolerant agent instance runs about 2.8% of the total current workload.

The following picture summarizes the workload balancing between the master and the fault-tolerant agents:

![Workload balancing between Master and fault-tolerant agents](image)

The following sections describe some characteristics of the workloads that are common for all the three levels (Light, Medium, and Heavy).

The last subsections, instead, give details about the characteristics that change from workload to workload, with explicit reference to the particular workload.

### 6.2.1 Jobs categories

The jobs to be run were split into 3 categories:

- Light jobs
- Medium jobs
Heavy jobs

A light job is a job having a very small impact on the machine that is running it, in terms of hardware resources consumption.

A light job simply runs the command “ls -la” in the working directory folder (list the detailed content of the current directory).

A medium job is a job having a medium impact on the machine that is running it, in terms of hardware resources consumption.

A medium job runs the command “ping” on the localhost machine 100 times, using the appropriate syntax ping command.

A heavy job is a job having a strong impact on the machine that is running it, in terms of hardware resource consumption.

A heavy job runs a script “heavyjob.sh” on the machine that loops for 5000 times without performing any operation, just to consume a non-negligible amount of hardware resources.

This is the heavyjob.sh script code (the same was used for both AIX and RHEL OS):

```bash
#!/myfile/heavyjob.sh

i=0
MAX=5000
while [ $i -lt $MAX ]
    do
        i=`expr $i + 1`
    done
```

Figure 11 - Heavy job script file

Depending on whether the agent is a fault-tolerant agent machine on the Linux RedHat operating system, or it is the Master, on an AIX operating system, the job command syntax can be slightly different, but what is actually run is the same.

The following table summarizes the job categories to be run in any workload of the Tivoli Workload Scheduler V8.5.1 engine Capacity Plan:

<table>
<thead>
<tr>
<th>job category</th>
<th>Impact on system</th>
<th>Command on RHEL OS</th>
<th>Command on AIX OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light job</td>
<td>light</td>
<td>ls -la</td>
<td>ls -la</td>
</tr>
<tr>
<td>Medium job</td>
<td>medium</td>
<td>ping localhost -c 100</td>
<td>ping -c 100 localhost</td>
</tr>
<tr>
<td>Heavy job</td>
<td>heavy</td>
<td>heavyjob.sh</td>
<td>heavyjob.sh</td>
</tr>
</tbody>
</table>

Table 5 - Jobs category details
For every workload, the 3 job categories are always divided with these percentages:

- Light job: 50% of the total number of jobs to be executed
- Medium job: 40% of the total number of jobs to be executed
- Heavy job: 10% of the total number of jobs to be executed

![Workload total jobs](image)

*Figure 12 - Job category percentages over the total number of jobs run in a given workload*

The job category is assigned to any job in a random way, compliant with those percentages (e.g. the probability that a job is assigned the “job light” category is 50%).

### 6.2.2 Jobs execution time

In the Tivoli Workload Scheduler V8.5.1 engine Capacity Plan, jobs have not a calendar rule to match to decide when they need to be executed.

The execution time is set at job streams level with the "AT" condition (expressed in hours and minutes) that allows to specify when the job streams must be executed.

In this way, the execution of any job inside a particular job stream will be triggered when:

- Current time on Master is equal to or greater than the time specified with the "AT" condition
- Job stream has satisfied any follows dependency constraints
- Job itself has satisfied any dependency constraints (follows, prompt, resources, files)
6.2.3 Job recovery policy

When a job fails its execution for any reason, the "recovery" syntax allows specifies the agent behavior to manage the failed job.

The possible options are:

- Stop => Stops the job leaving it in a failed status, blocking, as consequence, anything that depends on the correct completion of the job.
- Continue => Ignores the failed job, allowing anything that depends on the job to continue running
- Rerun => Runs the job again to try to complete it in a successful way

Regardless of the workload, the recovery policy is assigned in a random way, with the same percentage:

- Stop => 33% of the total number of jobs
- Continue => 33% of the total number of jobs
- Rerun => 33% of the total number of jobs

6.2.4 Job priority

Job priority sets a grade of importance to any job, so that the agent, in the case of having more than one job to potentially run, will run the one with the highest priority.

Job priority is assigned randomly, with the same probability, an integer in [1..99] interval.

6.2.5 Job dependencies and topology

In Tivoli Workload Scheduler V8.5.1 engine Capacity Plan, jobs have the following types of dependency:

- Follows dependencies between jobs belonging to the same job stream
- Resource dependencies on resources located on the same fault-tolerant agent instance running the job
- File dependencies on files located on the same fault-tolerant agent machine running the job
- Prompt dependencies from some predefined prompts.
The most significant fact is that follows dependencies exist only between jobs belonging to the same job stream. This means that the Capacity Plan does not cover cases of external follows dependencies at job level.

Considering the constraint that each job stream contains 20 jobs, a specific number of dependencies (of all the above categories) was defined for each job stream, so that the total number of Dependencies can be obtained by multiplying the number of job streams of the specific job streams for the number of job dependencies per job stream.

The following sections describe the dependencies of jobs and job streams in the workloads. It is important to note that all this information refers to any workload unless specified.

### 6.2.6 Prompt dependencies

For each job stream X of the current workload, the first three jobs (JOB_X_1, JOB_X_2 and JOB_X_3) have a prompt dependency on, respectively, three previously created prompt, named PRO_0, PRO_1, and PRO_2.

To allow the execution of that job, it is necessary for the operator to answer yes to the corresponding prompt. Until then, the job stays in hold, waiting for the prompt to be answered.

The following table summarizes the job prompt dependencies inside any job stream named X.

<table>
<thead>
<tr>
<th>Job name</th>
<th>Prompt dependency to prompt</th>
<th>Number of prompt dependencies per job</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_X_1</td>
<td>PRO_1</td>
<td>1</td>
</tr>
<tr>
<td>JOB_X_2</td>
<td>PRO_2</td>
<td>1</td>
</tr>
<tr>
<td>JOB_X_3</td>
<td>PRO_3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>3</strong></td>
</tr>
</tbody>
</table>

*Table 6 - Job prompt dependencies details for any job stream named x*

### 6.2.7 File dependencies

In each job stream named X, jobs numbered 4, 10, and 16 have a file dependency on some specific files. It is useful to note that “-f %p” flag was used so a file dependency is resolved when the agent can successfully check that the file exists and is a regular file.

Then, each of these jobs has a file dependency on one file, randomly picked, with the same probability, from this set = {"/myfiles/test0.sh", "/myfiles/test1.sh", "/myfiles/test2.sh", "/myfiles/test3"}. This means that each machine in the test environment must have, as a test prerequisite, those files existing in their file system, otherwise the dependencies cannot be resolved.

The following table summarizes the file dependencies for jobs inside any job stream X.
### Workload

<table>
<thead>
<tr>
<th>Job name</th>
<th>File dependency</th>
<th>File deps per job</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_X_4</td>
<td>1 file randomly picked in {&quot;/myfiles/test0&quot;, &quot;/myfiles/test1&quot;, &quot;/myfiles/test2&quot;, &quot;/myfiles/test3&quot;}</td>
<td>1</td>
</tr>
<tr>
<td>JOB_X_10</td>
<td>1 file randomly picked in {&quot;/myfiles/test0&quot;, &quot;/myfiles/test1&quot;, &quot;/myfiles/test2&quot;, &quot;/myfiles/test3&quot;}</td>
<td>1</td>
</tr>
<tr>
<td>JOB_X_16</td>
<td>1 file randomly picked in {&quot;/myfiles/test0&quot;, &quot;/myfiles/test1&quot;, &quot;/myfiles/test2&quot;, &quot;/myfiles/test3&quot;}</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>3</strong></td>
</tr>
</tbody>
</table>

*Table 7 - Job file dependency details for any job stream named x*

#### 6.2.8 Resource dependencies

On each fault-tolerant agent machine that is hosting the four fault-tolerant agent instances, and on the engine server, 20 distinct resources were created, with this naming convention: RES_1, RES_2, ..., RES_20.

Each resource has a quantity of 1024. This means that, in total, there are (8 fault-tolerant agent machines * 4 fault-tolerant agent instances) * 20 resources + Master *20 resources = 640 + 20 = 660 resources.

Job streams named X, JOB_X_6, JOB_X_12 and JOB_X_18 have a resource dependency on 3 different resources. For a given fault-tolerant agent machine, the first job stream will have JOB_1_6 that needs the resource RES_1, JOB_1_12 that needs the resource RES_2, and JOB_1_18 that needs the resource RES_3.

The first job of the next job stream belonging to the same fault-tolerant agent machine that needs the resource (JOB_2_6), will have a resource dependency on RES_4, and so on.

When a certain job stream’s job has a resource dependency on RES_20, the next job that needs a resource will have a resource dependency on RES_1, in a cyclic way.

The following table summarizes the job dependencies toward resources, for any generic job stream X:

<table>
<thead>
<tr>
<th>job name</th>
<th>Resource dependency</th>
<th>Res deps per job</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_X_6</td>
<td>RES_i</td>
<td>1</td>
</tr>
<tr>
<td>JOB_X_12</td>
<td>RES_i+1</td>
<td>1</td>
</tr>
<tr>
<td>JOB_X_18</td>
<td>RES_i+2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>3</strong></td>
</tr>
</tbody>
</table>

*Table 8 - Resource dependencies for jobs of generic job streams named x*
6.2.9 Light workload topology and dependencies

This section describes the topology and the dependencies used in the Light workload. The main characteristics of the Light workload, in terms of number of jobs, job streams, and dependencies are displayed in the following table:

<table>
<thead>
<tr>
<th>Workload</th>
<th>Jobs</th>
<th>Job streams</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>WL</td>
<td>100,000</td>
<td>5,000</td>
<td>200,000</td>
</tr>
</tbody>
</table>

Table 9 - Light workload details

It is useful to remember that, for each workload, there are 20 jobs for each job stream.

To obtain 200,000 dependencies, a specific job dependency topology was created so that each job stream contains 39 dependencies at job level, distributed as follows:

- 30 follows dependencies
- 3 prompt dependencies on predefined prompts
- 3 resource dependencies
- 3 file dependencies.

At this point, add the number of job streams follow dependencies that can be obtained as \( \frac{2}{3} \times \) number of job streams. This number comes from the particular topology: each job stream in 1,3,5...2i + 1 sequence, has 2 “follows” arcs towards the 2 following job streams. This means that each 3 consecutive job streams, starting from the first one, there are 2 dependency “follows” arcs; then, the total number of job streams “follows” dependency can be obtained as \( \frac{2}{3} \times \) number of job streams.

In this way, with the total number of 5,000 job streams, and each single job stream having 39 dependencies, a value of about 195,000 jobs dependencies (follows, prompts, resources, files) can be obtained to give a total number of:

195,000 dependencies at job level + 3,333 follows dependencies at job stream level = 198,333 dependencies

The following table summarizes the dependency types and amounts:

<table>
<thead>
<tr>
<th>Job follows dep/JS</th>
<th>Job resource dep/JS</th>
<th>Job prompt dep/JS</th>
<th>Job file dep/JS</th>
<th>Number of JS</th>
<th>JS follows dep</th>
<th>Total number of dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5,000</td>
<td>3,333</td>
<td>198,333</td>
</tr>
</tbody>
</table>

Table 10 - Resource types and amounts for Light workload

Follows dependencies were created with the following rule: for any job stream, each job (numbered starting from 1) in the number sequence 1,4,7,10...19 \((1 + 3x, \text{being } x \text{ an integer in interval } [0..6])\) has a follows dependency towards the next 5 jobs (ordered by number).

Of course, jobs 16 and 19 will have, respectively, only 4 and 1 follows dependencies.
The following table enumerates, for any given job stream named x, the jobs having follows dependencies.

<table>
<thead>
<tr>
<th>job name</th>
<th>Depends on</th>
<th>Follows deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_x_1</td>
<td>JOB_x_2, JOB_x_3, JOB_x_4, JOB_x_5, JOB_x_6</td>
<td>5</td>
</tr>
<tr>
<td>JOB_x_4</td>
<td>JOB_x_5, JOB_x_6, JOB_x_7, JOB_x_8, JOB_x_9</td>
<td>5</td>
</tr>
<tr>
<td>JOB_x_7</td>
<td>JOB_x_8, JOB_x_9, JOB_x_10, JOB_x_11, JOB_x_12</td>
<td>5</td>
</tr>
<tr>
<td>JOB_x_10</td>
<td>JOB_x_11, JOB_x_12, JOB_x_13, JOB_x_14, JOB_x_15</td>
<td>5</td>
</tr>
<tr>
<td>JOB_x_13</td>
<td>JOB_x_14, JOB_x_15, JOB_x_16, JOB_x_17, JOB_x_18</td>
<td>5</td>
</tr>
<tr>
<td>JOB_x_16</td>
<td>JOB_x_17, JOB_x_18, JOB_x_19, JOB_x_20</td>
<td>4</td>
</tr>
<tr>
<td>JOB_x_19</td>
<td>JOB_x_20</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

Table 11 – Light workload: job follows dependency topology for a given job streams named x

The following scheme summarizes, in a graphical way, the topology of the dependencies of the jobs in a generic job stream named x, for the Light workload:

![Figure 13 - Light workload: job dependencies graphical description for a generic job stream named x](image-url)
6.2.10 Medium workload topology and dependencies

This section describes the topology and the dependencies used in the Medium workload. The main characteristics of the Medium workload, in terms of number of jobs, job streams, and dependencies are shown in the following table:

<table>
<thead>
<tr>
<th>Workload</th>
<th>Jobs</th>
<th>Job streams</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>WM</td>
<td>200,000</td>
<td>10,000</td>
<td>250,000</td>
</tr>
</tbody>
</table>

*Table 12 - Medium workload details*

To obtain 250,000 dependencies, a specific job dependency topology was created so that each job stream contains 24 dependencies at job level, distributed as follows:

- 15 follows dependencies
- 3 prompt dependencies
- 3 resource dependencies
- 3 file dependencies.

Then, considering the total number of 10,000 job streams, with each job stream having 24 job dependencies, there will be a total amount of 24 * 10,000 = 240,000 job dependencies (follows, resources, files, prompts).

Recalling the fact that, for the job stream topology detailed below, there are also 2/3 * number of job streams of follows dependencies between job streams, the resulting total number of dependencies for the Medium workload is:

240,000 job dependencies (follows, resources, prompts, files) + 6,666 job streams dependencies (follows) = 246,666 dependencies.

Following table summarizes the resources types and amounts, for the Medium workload:

<table>
<thead>
<tr>
<th>Job follows dep/JS</th>
<th>Job resource dep/JS</th>
<th>Job prompt dep/JS</th>
<th>Job file dep/JS</th>
<th>Number of JS</th>
<th>JS follows dep</th>
<th>Total number of dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>10,000</td>
<td>6,666</td>
<td>246,666</td>
</tr>
</tbody>
</table>

*Table 13 - Resource types and amounts for Medium workload*

Follows dependencies were created with the following rule: for any job stream, each job (numbered starting from 1) in the number sequence 1,4,7,10…19 (1 + 3x, x being an integer in interval [0..6]) has a follows dependency towards the next 2 jobs (ordered by number).

The only exceptions are for:

- JOB_x_1: it has a follows dependency also towards 3rd and 4th consecutive job
- **JOB_x_19**: it has only a single follows dependency towards the last job JOB_X_20

The following table enumerates, for any given job streams named x, the jobs having follows dependencies:

<table>
<thead>
<tr>
<th>job name</th>
<th>follows dependency to jobs</th>
<th>Follows deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_x_1</td>
<td>JOB_x_2, JOB_x_3, JOB_x_4, JOB_x_5</td>
<td>4</td>
</tr>
<tr>
<td>JOB_x_4</td>
<td>JOB_x_5, JOB_x_6</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_7</td>
<td>JOB_x_8, JOB_x_9</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_10</td>
<td>JOB_x_11, JOB_x_12</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_13</td>
<td>JOB_x_14, JOB_x_15</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_16</td>
<td>JOB_x_17, JOB_x_18</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_19</td>
<td>JOB_x_20</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>15</strong></td>
<td></td>
</tr>
</tbody>
</table>

*Table 14 – Medium workload: job follows dependency topology for a given job stream named x*

The following scheme summarizes, in a graphical way, the topology of the dependencies of the jobs a generic job stream named x, for the Medium workload:

*Figure 14 - Medium workload: jobs dependency graphical description for a generic job stream named x*
6.2.11 Heavy workload topology and dependencies

This section describes the topology and the dependencies used in the Heavy workload. The main features of the Heavy workload, in terms of number of jobs, job streams, and dependencies are displayed in the following table:

<table>
<thead>
<tr>
<th>Workload name</th>
<th>Jobs</th>
<th>Job streams</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>WH</td>
<td>300,000</td>
<td>15,000</td>
<td>300,000</td>
</tr>
</tbody>
</table>

*Table 15 - Heavy workload details*

In order to obtain 30,000 dependencies, a specific job dependency topology was created so that each job stream contains 20 dependencies, distributed in this way:

- 11 follows dependencies between jobs belonging to the current job stream
- 3 prompt dependencies on some defined prompts
- 3 resource dependencies on some defined resources
- 3 file dependencies on some created files.

Then, considering the total number of 15000 job streams, each job stream having a 20 job dependencies, there will be a total amount of 20 * 15,000 = 300,000 job dependencies (follows, resources, files, prompts).

Recalling the fact that, for the job stream topology detailed below, there are also 2/3 * number of job streams of follows dependencies between job streams, the resulting total number of dependencies for the Heavy workload is:

300,000 job dependencies (follows, resources, prompts, files) + 10,000 job streams dependencies (follows) = 310,000 dependencies.

The following table summarizes the resources types and amounts, for the Heavy workload:

<table>
<thead>
<tr>
<th>Job follows dep/JS</th>
<th>Job resource dep/JS</th>
<th>Job prompt dep/JS</th>
<th>Job file dep/JS</th>
<th>Number of JS</th>
<th>JS follows dep</th>
<th>Total number of dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>15,000</td>
<td>10,000</td>
<td>310,000</td>
</tr>
</tbody>
</table>

*Table 16 - Resource types and amounts for Heavy workload*

Follows dependencies were created with the following rule: for any job stream, each job (numbered starting from 1) in the number sequence 1,4,7,10…19 (1 + 3x, x being an integer in interval [0..6]) has a follows dependency towards:

- The first 2 consecutive jobs (for jobs 1,4,7,10)
- The first single consecutive job (for job 13,16,19)
The following table enumerates, for any given job stream named x, the jobs having follows dependencies:

<table>
<thead>
<tr>
<th>job name</th>
<th>follows dependency to jobs</th>
<th>Follows deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_x_1</td>
<td>JOB_x_2, JOB_x_3</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_4</td>
<td>JOB_x_5, JOB_x_6</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_7</td>
<td>JOB_x_8, JOB_x_9</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_10</td>
<td>JOB_x_11, JOB_x_12</td>
<td>2</td>
</tr>
<tr>
<td>JOB_x_13</td>
<td>JOB_x_14</td>
<td>1</td>
</tr>
<tr>
<td>JOB_x_16</td>
<td>JOB_x_17</td>
<td>1</td>
</tr>
<tr>
<td>JOB_x_19</td>
<td>JOB_x_20</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>11</strong></td>
</tr>
</tbody>
</table>

*Table 17 - Heavy workload: job follows dependency topology for a given job streams named x*

The following scheme summarizes, in a graphical way, the topology of the dependencies of the jobs in a generic job stream named x, for the Heavy workload:

*Figure 15 - Heavy workload: jobs dependency graphical description for a generic job stream named x*
6.2.12 Job stream properties: “AT”

Every job stream of the Tivoli Workload Scheduler V8.5.1 engine Capacity Plan contains an “AT” condition, to specify, in conjunction with the “ON EVERYDAY” condition, to run everyday of the week the jobs contained in that job stream in a server time equal to the time specified by the four digits after the “AT”.

“AT” time of the day values are generated in a random way, with equal probability for any time value to be picked (uniform distribution of values), inside a certain specified time interval, to have job execution only in a slot time of the day.

For the Tivoli Workload Scheduler V8.5.1 engine Capacity Plan, the AT interval was set from 1200 to 1700 (12.00 pm to 5.00 pm, with a length of 5 hours); this way, each job stream has an execution time AT from 1200 to 1700, with the same probability of being randomly chosen.

6.2.13 Job stream properties: carryforward

Each job stream has a 50% probability to have the “carryforward” set, and a 50% probability to have the “carryforward” not set.

This is done randomly for each job stream.

6.2.14 Job stream properties: priority

As for the job, the priority for job streams is given picking randomly, with the same probability, an integer in [0..99] interval.

6.2.15 Job stream properties: limit

Limit is set to each job stream picking randomly, with the same probability, an integer value inside the [1..1024] interval.

6.2.16 Job stream properties: dependencies and topology

In Tivoli Workload Scheduler V8.5.1 engine Capacity Plan, job streams have only follows dependencies. In particular follows dependencies were set only between job streams belonging to the same fault-tolerant agent instance installed on a specific fault-tolerant agent machine, compliant with a rule based on the enumeration.

With a given fault-tolerant agent instance installed on a given fault-tolerant agent machine, each job stream with a number in the series 1, 4, 7, … (1 +3x, x being an integer greater than 0 and limited to the number of job streams of the current workload), has 2 follows dependencies, compliant with this rule:

1. A follows dependency towards the first job stream with increased number compared with the current
2. A follows dependency towards the second job stream with increased number compared with the current.

The following figure displays the job streams follows dependencies for a given fault-tolerant agent instance, installed on a given fault-tolerant agent machine:

![Figure 16 - Job stream "follows "dependencies in a given fault-tolerant agent instance on a specific fault-tolerant agent i machine](image)

It is important to note that the same job stream follows dependency topology was used for all the three workload levels (Light, Medium and, Heavy), with the only difference being the total number of job streams.

Then, as a consequence of this job stream dependencies topology, it is possible to assert that:

- Job streams named JS_i, i being in the sequence 1,4,7...(1 + 3x), can only be run when both the job streams that it is in follows dependency with go into success status (that is, every job contained in both of them must successfully complete the execution), and when the "AT hhmm" condition is respected (current time is equal or greater than that specified time)

- Job streams named JS_j, j being in the sequence 2,5,8 (2 + 3x), can only be run when the "AT hhmm" condition is respected (no follows dependency with other job streams)
6.3 Dynamic job submission workload

This section describes the workload used to run the dynamic job submission. As described above, the dynamic job submission test activities were run using a different workload, compared with the one used, in the 3 workload levels versions, for the plan preparation, execution, and further tests activities.

This workload is mainly composed of two components:

- Scheduling objects making up the plan
- Scheduling objects making the jobs to be submitted

It is important to remember that tests were executed in 2 distinct modes:

- Job submission during a plan execution
- Job submission with no plan execution

This means that the scheduling activity only affects the job submission in the first way, which is when the plan is actually executed while submitting jobs.

Tests were run to highlight how the “plan execution”, considerable in this case as a sort of “background noise” in terms of resource consumption for the engine, can affect the job submission.

The job submission is run by a script running on the engine server itself, which submits jobs inside an infinite loop.

6.3.1 Plan scheduling objects

To drill down, into more detail the workload used for the plan part was obtained by simplifying the one used for the plan preparation and execution activities, by reducing the number of jobs, and removing the follows dependencies between jobs in the same job stream and distinct job streams, obtaining:

- 50,000 jobs packed in 5,000 job streams, each containing 10 jobs
- No follows dependencies between jobs belonging to the same job stream
- No follows dependencies between distinct job streams
- Same other dependencies as the plan preparation and execution workloads (files, prompts, and resources dependencies), to the same objects (files existing on fault-tolerant agents and engine, defined resources for each fault-tolerant agent and the engine, defined prompts)
• Same jobs typology as the plan preparation and execution workloads (light, medium, and heavy job types)

As for the plan execution and preparation tests, the workload for the plan part was balanced as follows:

• 90% of the total workload to be run by the fault-tolerant agents (about 4500 job streams each containing 10 jobs)

• 10% of the total workload to be run by the Master (500 job streams each containing 10 jobs)

The following section highlights the differences with the previous workload used in the plan preparation and execution.

Because there are no follows dependencies between job streams, each job stream will be simply launched by the engine as soon as the scheduled start time “AT” matches.

For each job stream, some jobs have dependency to file, resource or prompt, in this way:

• First job has a dependency on a resource

• Second job has a dependency on a prompt

• Third job has a dependency on a file

In this way, there are 15000 dependencies, equally split into file, prompt, and resource dependencies, obtained by multiplying 3 (number of dependencies per job stream) for 5000 (number of job streams).

<table>
<thead>
<tr>
<th>Jobs</th>
<th>Job streams</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>5000</td>
<td>15000</td>
</tr>
</tbody>
</table>

*Figure 17 - Dynamic job submission workload description*

The following picture summarizes the entire workload used for the dynamic jobs submission activities:
6.3.2 Job submission scheduling objects

This section describes the scheduling object structure for the actual submission part of the test activity.

The jobs to be submitted are 10, inserted in 10 distinct job streams, with the following features:

- Jobs in the same job stream build a “follows dependency chain” from the first to the last job in each job stream
- No AT syntax is present at any level, so that jobs are submitted immediately.
- Job type is, for all the jobs, the same, that is a light job

As for the plan section, the jobs submission workload part was balanced in this way:

- 90% of the workload run by fault-tolerant agent (in this case, all by the first instance fault-tolerant agent_023_0)
- 10% of the workload run by the Master
The job submission was performed using a script that, in an infinite loop, keeps submitting the 10 job streams to the engine, with no artificial pause between each single submission (so the submission frequency/speed depends on the currently available engine resources, during the test execution).

The following picture shows the job submission workload part.

![Figure 19 - Dynamic job submission: jobs to submit details](image-url)
7 Conclusions and recommendations

Tivoli Workload Scheduler V8.5.1 engine capacity plan test activities were all successfully executed and completed.

It was possible to execute and complete, from a functional point of view, each planned test:

- Plan creation
- Plan execution
- Plan extension
- Dynamic job submission

From a performance point of view, the next section provides details about the most suitable configuration to use, for the Tivoli Workload Scheduler engine server, to obtain best performances for a particular feature, such as CPU usage, RAM, disk, or network activities, throughput or delay in tasks completion.

In more details, the next section proposes recommendations and tables to easily identify, for a given workload, the most suitable configuration (or discriminating configuration feature, such as amount of RAM or number of CPUs) that get best performances for a particular server feature.

In this way, a customer needing to identify, for their own input workload, which is the most suitable configuration to obtain best performances, can get answers by consulting this document, identifying the most similar analyzed test scenario with their required one, and getting information about how to configure the server for best performance in that particular scenario.

7.1 JnextPlan – Recommendations

This section summarizes all the recommendations provided for the analyzed server features and resources, for the different workloads, and for the different configurations, for the JnextPlan test activities.

Recommendations are provided in a table, to make it easier to decide which configuration to choose.

Table 18 shows the best configuration to choose to optimize a particular resource usage. If there is more than one configuration that represents the optimum, they are listed together, highlighting any common features of the optimizing configurations.

If there is no particular configuration optimizing the resource usage, “Any config” is reported so that any configuration can be chosen.
## 7.2 Execution – Recommendations

This section summarizes all the recommendations provided for the analyzed server features and resources, for the different workloads and for the different configurations, for the execution testing activities.

Recommendations are provided in a table, to make it easier to decide which configuration to choose.

Table 19 shows the best configuration to choose to optimize a particular resource usage. If there is more than one configuration that represents the optimum, they are listed together, highlighting any common features of the optimizing configurations.

If there is no particular configuration optimizing the resource usage, “Any config” is reported so that any configuration can be chosen.
7.3 Extension – Recommendations

The extension results analysis highlighted the following items:

- JnextPlan from scratch, for any given configuration, and any given workload, takes about the same time to complete, after several trials.

- JnextPlan extensions when Medium Workload is submitted can successfully be completed only with configurations provided with 4 GB of RAM (Config3 and Config5); this is a system breaking point.

- JnextPlan extensions when Heavy Workload is submitted cannot be completed in any configuration, always failing for “OutOfMemory” exception; this means that a 4 GB configuration should be selected, and there is the need to increase the heap size to the limit; in this case, initial heap could be set to 1.5 GB (1536 MB), while maximum heap could be set to 1.7 GB (1768 MB); this could solve the problem of out of memory exceptions; this is a system breaking point.

The following table summarizes the configuration to select to obtain best performances on features.

![Table 19 - Execution - recommendations for best performances](image)

![Table 20 - Extension - Recommendations for best performances on several features](image)
7.4 Dynamic – Recommendations

This section summarizes the dynamic test activities results analysis providing a table to help the reader to address and choose the correct Tivoli Workload Scheduler engine server configuration, considering the input workload and any other constraint, to get best performance for this kind of, or a similar activity (dynamic job submission).

<table>
<thead>
<tr>
<th>Best perf Config.</th>
<th>CPU usage</th>
<th>Disk read</th>
<th>Disk write</th>
<th>Memory usage</th>
<th>Network usage</th>
<th>Job submission rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>No plan</td>
<td>Cfg4, Cfg5</td>
<td>Cfg3</td>
<td>Any config</td>
<td>Cfg1, Cfg2</td>
<td>Any config</td>
<td>Cfg5</td>
</tr>
<tr>
<td>Plan</td>
<td>Cfg4, Cfg5</td>
<td>Cfg3</td>
<td>Cfg2, Cfg4, Cfg5</td>
<td>Cfg2, Cfg3, Cfg4</td>
<td>Cfg2, Cfg4, Cfg5</td>
<td>Cfg2, Cfg4, Cfg5</td>
</tr>
</tbody>
</table>

*Table 21 - Dynamic job submission, recommendations*

7.5 Other suggestions and tuning

Tuning and settings information is also provided, to optimize the server usage to get the best performances.

Details are provided in the related sections below, but are also be summarized in this list:

- Database tuning (details in the relevant section)
- WAS heap size tuning (details in the relevant section)
- Communication queues settings (details in the relevant section)
- Full status configuration (details in the opportune section)

In particular, database and heap configuration are listed for all the environments tested to tune the system for best performance.

Further features found during the test execution can be identified as:

- Communication queues between agents and engine can go to saturation if the full status is enabled, under the heavy workload.
- System breaking points in the plan extension for medium and heavy workloads.
The first issue was addressed by switching off the full status feature, leading to an important performance improvement (and to the test completing successfully).

The second issue highlighted the fact that, for the Heavy Workload, a breaking point was found.

Considering all the features and factors, this test analysis has demonstrated that the Tivoli Workload Scheduler engine server can manage all the provided workloads, successfully completing each test from a functional point of view, and offering very interesting tuning settings and tips on how to obtain best performance for many different features that the customer might be interested in.
IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.