

HPSA Extension Pack

Equipment Connection Pool

Release V6.0



Legal Notices

Warranty.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

A copy of the specific warranty terms applicable to your Hewlett-Packard product can be obtained from your local Sales and Service Office.

Restricted Rights Legend.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company United States of America

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Copyright Notices.

©Copyright 2001-2012 Hewlett-Packard Development Company, L.P., all rights reserved.

No part of this document may be copied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this material is subject to change without notice.

Trademark Notices.

Java™ is a trademark of Oracle and/or its affiliates.

Linux is a U.S. registered trademark of Linus Torvalds

Microsoft® is a U.S. registered trademark of Microsoft Corporation.

Oracle® is a trademark of Oracle and/or its affiliates.

UNIX® is a registered trademark of the Open Group.

Windows® and MS Windows® are U.S. registered trademarks of Microsoft Corporation.

All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

Document id: EP-pd001801

Table of Contents

| | |
|--|----|
| Legal Notices..... | 2 |
| Table of Contents | 3 |
| Audience..... | 7 |
| Install Location Descriptors..... | 9 |
| 1. Introduction | 11 |
| 1.1. Purpose..... | 11 |
| 1.2. General Description | 11 |
| 1.3. ECP Module Entities and Concepts | 12 |
| 1.3.1. Target System..... | 12 |
| 1.3.2. Operation..... | 12 |
| 1.3.3. Commands Template/Operation Template | 12 |
| 1.3.4. Operation Execution | 13 |
| 1.3.5. Resource | 13 |
| 1.3.6. Pool | 13 |
| 1.3.7. SubPool..... | 13 |
| 1.3.8. Equipment Driver | 13 |
| 1.3.9. Protocol Driver | 14 |
| 1.3.10. ECP Instance | 14 |
| 1.4. General Architecture | 14 |
| 1.4.1. ECP Client | 14 |
| 1.4.2. ECP Service..... | 15 |
| 2. Functionality and Architecture | 17 |
| 2.1. Connection and Pool Management..... | 17 |
| 2.1.1. Connection Reuse..... | 17 |
| 2.1.2. High Availability..... | 17 |
| 2.1.3. Target System Independence | 18 |
| 2.1.4. Protocol Independence | 18 |
| 2.1.5. Load Balance | 19 |
| 2.2. Pool and Connection types..... | 19 |
| 2.2.1. Static vs Temporary Pools..... | 19 |
| 2.2.2. Direct Connections (Not Pooled Connections) | 19 |
| 2.2.3. Dynamic Pools | 19 |
| 2.3. Commands Template..... | 20 |
| 2.4. Operation Execution | 20 |
| 2.5. Real-time Monitoring | 22 |
| 3. First Steps..... | 23 |
| 3.1. Equipment Driver Development..... | 23 |
| 3.1.1. Equipment Driver Development Introduction..... | 23 |
| 3.1.2. Equipment Driver Generic | 25 |
| 3.1.3. Equipment Driver Deployment..... | 26 |
| 3.1.4. Available Equipment Drivers..... | 26 |

Equipment Connection Pool

| | | |
|--------|---|----|
| 3.1.5. | Generic Template Equipment Driver..... | 27 |
| 3.2. | ECP Service Process | 30 |
| 3.2.1. | Starting ECP Service..... | 30 |
| 3.2.2. | Stopping ECP Service | 30 |
| 3.2.3. | Restarting ECP Service | 30 |
| 3.2.4. | Checking ECP Service..... | 30 |
| 3.3. | Use Examples..... | 31 |
| 3.3.1. | Creating and Using an Static Pool | 32 |
| 3.3.2. | Creating and Using a Dynamic Pool..... | 35 |
| 3.3.3. | Using Direct Connections | 36 |
| 3.4. | Monitoring ECP through JMS | 37 |
| 3.4.1. | Including Additional Data in Activation JMS Messages: | 37 |
| 3.4.2. | JMS Client Dependencies | 38 |
| 3.4.3. | JMS Client Examples | 39 |
| 3.4.4. | ECP Messages Types | 43 |
| 3.5. | Retrieving connection information through an external data getter | 44 |
| 4. | Configuration..... | 45 |
| 4.1. | Common Configuration Sources | 45 |
| 4.1.1. | ProtocolDrivers.lst File | 45 |
| 4.1.2. | Protocol Drivers Configuration | 45 |
| 4.1.3. | HPSA_ECPMESSAGESPATTERNS | 46 |
| 4.1.4. | HPSA_ECPCOMMANDSPATTERNS | 46 |
| 4.1.5. | HPSA_ECPMESSAGESCOMMANDS | 47 |
| 4.2. | ECP Lib Configuration Sources | 47 |
| 4.2.1. | ECP Lib Command Line Parameters..... | 47 |
| 4.3. | ECP RMI Service Configuration Sources | 47 |
| 4.3.1. | ECP RMI Service Command Line Parameters | 47 |
| 4.3.2. | ecp-cluster.properties | 50 |
| 4.3.3. | HPSA_EQUIPMENTCONNPOOL DB Table | 50 |
| 4.3.4. | SubPool Instance Specific Parameters Configuration | 52 |
| 5. | Commands Template Reference | 54 |
| 5.1. | Commands Template Commands | 54 |
| 5.1.1. | Block declaration Statements..... | 54 |
| 5.1.2. | Executable Statements..... | 54 |
| 5.1.3. | Command Statements | 55 |
| 5.2. | Commands Reference | 55 |
| 5.2.1. | Commands List | 55 |
| 5.2.2. | Commands Syntax | 57 |
| 6. | Configuration Quick Reference | 60 |
| 6.1. | DBManager Configuration | 60 |
| 6.2. | Configurator Configuration | 60 |
| 6.3. | ECP RMI Service | 60 |
| 6.4. | PoolManager Configuration..... | 61 |
| 6.5. | Pool Configuration | 61 |
| 6.5.1. | Pool Common Parameters Configuration..... | 61 |
| 6.5.2. | Pool Instance Specific Parameters Configuration..... | 62 |
| 6.6. | SubPool Configuration | 64 |
| 6.6.1. | SubPool Instance Specific Parameters Configuration | 64 |

| | | |
|--------|--|----|
| 6.7. | EquipmenDriver Configuration..... | 67 |
| 6.7.1. | EquipmentDriver Initialization Parameters Configuration..... | 67 |
| 6.7.2. | ConnectionResource Configuration | 68 |
| 6.8. | Protocol Drivers Manager Configuration..... | 68 |
| 6.9. | ProtocolDriver Configuration..... | 68 |
| 6.10. | CLICommands Configuration | 68 |
| 6.11. | Template Parser Configuration..... | 68 |
| 6.12. | JMS Monitoring Configuration..... | 69 |

In This Guide

This guide will explain the configuration, installation, needed development, and functionality provided by the ECP.

Audience

The audience for this guide is the Solutions Integrator (SI). The SI has a combination of some or all of the following capabilities:

Understands and has a solid working knowledge of:

- UNIX® commands

- Windows® system administration

Understands networking concepts and language

Is able to program in Java™ and XML

Understands security issues

Understands the customer's problem domain

Equipment Connection Pool

Conventions

The following typographical conventions are used in this guide.

| Font | What the Font Represents | Example |
|----------------------|---|--|
| <i>Italic</i> | Book or manual titles, and man page names | Refer to the <i>HP Service Activator — Workflows and the Workflow Manager</i> and the <i>Javadocs</i> man page for more information. |
| | Provides emphasis | You <i>must</i> follow these steps. |
| | Specifies a variable that you must supply when entering a command | Run the command: <code>java -classpath <classpath></code> |
| | Parameters to a method | The <i>assigned_criteria</i> parameter returns an ACSE response. |
| Bold | New terms | The distinguishing attribute of this class... |
| Computer | Text and items on the computer screen | The system replies: <code>Press Enter</code> |
| | Command names | Use the <code>java</code> command ... |
| | Method names | The <code>get_all_replies()</code> method does the following... |
| | File and directory names | Edit the file <code>\$ACTIVATOR_ETC/config/mwfm.xml</code> |
| | Process names | Check to see if <code>mwfm</code> is running. |
| | Properties files keys names | Set the property <code>LOG_DIR</code> to establish the log files path. |
| | Window/dialog box names | In the <code>Test and Track</code> dialog... |
| | XML tag references | Use the <code><DBTable></code> tag to... |
| Computer Bold | Text that you must type | At the prompt, type: <code>ls -l</code> |
| Keycap | Keyboard keys | Press Return . |
| [Button] | Buttons on the user interface | Click [Delete]. Click the [Apply] button. |
| Menu Items | A menu name followed by a colon (:) means that you select the menu, then the item. When the item is followed by an arrow (->), a cascading menu follows | Select <code>Locate:Objects->by Comment</code> . |

Install Location Descriptors

The following names are used throughout this guide to define install locations.

| Descriptor | What the Descriptor Represents |
|----------------------|---|
| \$ACTIVATOR_OPT | The base install location of Service Activator. The UNIX location is /opt/OV/ServiceActivator The Windows location is <drive>:\HP\OpenView\ServiceActivator\ |
| \$ACTIVATOR_ETC | The install location of specific Service Activator configuration files. The UNIX location is /etc/opt/OV/ServiceActivator The Windows location is <drive>:\HP\OpenView\ServiceActivator\etc\ |
| \$ACTIVATOR_VAR | The install location of specific Service Activator logging files. The UNIX location is /var/opt/OV/ServiceActivator The Windows location is <drive>:\HP\OpenView\ServiceActivator\var\ |
| \$ACTIVATOR_BIN | The install location of specific Service Activator binary files. The UNIX location is /opt/OV/ServiceActivator/bin The Windows location is <drive>:\HP\OpenView\ServiceActivator\bin\ |
| \$JBOSS_HOME | HOME The install location for JBoss. The UNIX location is /opt/HP/jboss The Windows location is <drive>:\HP\jboss |
| \$JBOSS_DEPLOY | The install location of the Service Activator J2EE components. The UNIX location is /opt/HP/jboss/standalone/deployments The Windows location is <drive>:\HP\jboss\stanalone\deployments |
| \$ACTIVATOR_DB_USER | The database user name you define. Suggestion: ovactivator |
| \$ACTIVATOR_SSH_USER | The Secure Shell user name you define. Suggestion: ovactusr |
| \$SOSA_HOME | The base install location of SOSA. The UNIX location is /opt/OV/Sosa The Windows location is <drive>:\HP\OpenView\Sosa\ |
| \$SOSA_BIN | The install location of specific SOSA binary files. |

Equipment Connection Pool

| | |
|------------|--|
| | <p>The UNIX location is /opt/OV/ServiceActivator/EP/SOSA/bin</p> <p>The Windows location is</p> <p><drive>:\HP\OpenView\ServiceActivator\EP\SOSA\bin\</p> |
| \$SOSA_ETC | <p>The install location of specific SOSA configuration files.</p> <p>The UNIX location is</p> <p>/opt/OV/ServiceActivator/EP/SOSA/conf</p> <p>The Windows location is</p> <p><drive>:\HP\OpenView\ServiceActivator\EP\SOSA\conf\</p> |
| \$ECP_HOME | <p>The base install location of Equipment Connections Pool.</p> <p>The UNIX location is /opt/OV/ServiceActivator/EP/ECP</p> <p>The Windows location is</p> <p><drive>:\HP\OpenView\ServiceActivator\EP\ECP\</p> |
| \$ECP_BIN | <p>The install location of specific Equipment Connections Pool binary files.</p> <p>The UNIX location is /opt/OV/ServiceActivator/EP/ECP/bin</p> <p>The Windows location is</p> <p><drive>:\HP\OpenView\ServiceActivator\EP\ECP\bin\</p> |
| \$ECP_ETC | <p>The install location of specific Equipment Connections Pool configuration files.</p> <p>The UNIX location is /opt/OV/ServiceActivator/EP/ECP/conf</p> <p>The Windows location is</p> <p><drive>:\HP\OpenView\ServiceActivator\EP\ECP\conf\</p> |
| \$ECP_LIB | <p>The install location of specific Equipment Connections Pool jar files.</p> <p>The UNIX location is /opt/OV/ServiceActivator/EP/ECP/lib</p> <p>The Windows location is</p> <p><drive>:\HP\OpenView\ServiceActivator\EP\ECP\lib\</p> |
| \$ECP_LOG | <p>The install location of specific Equipment Connections Pool log files.</p> <p>The default UNIX location is</p> <p>/opt/OV/ServiceActivator/EP/ECP/log</p> <p>The default Windows location is</p> <p><drive>:\HP\OpenView\ServiceActivator\EP\ECP\log\</p> |

1. Introduction

1.1. Purpose

This document is a manual for all ECP Module users. It gives a general view of the ECP Module concepts, functionality, architecture, and use, with special focus in configuration and its effects.

1.2. General Description

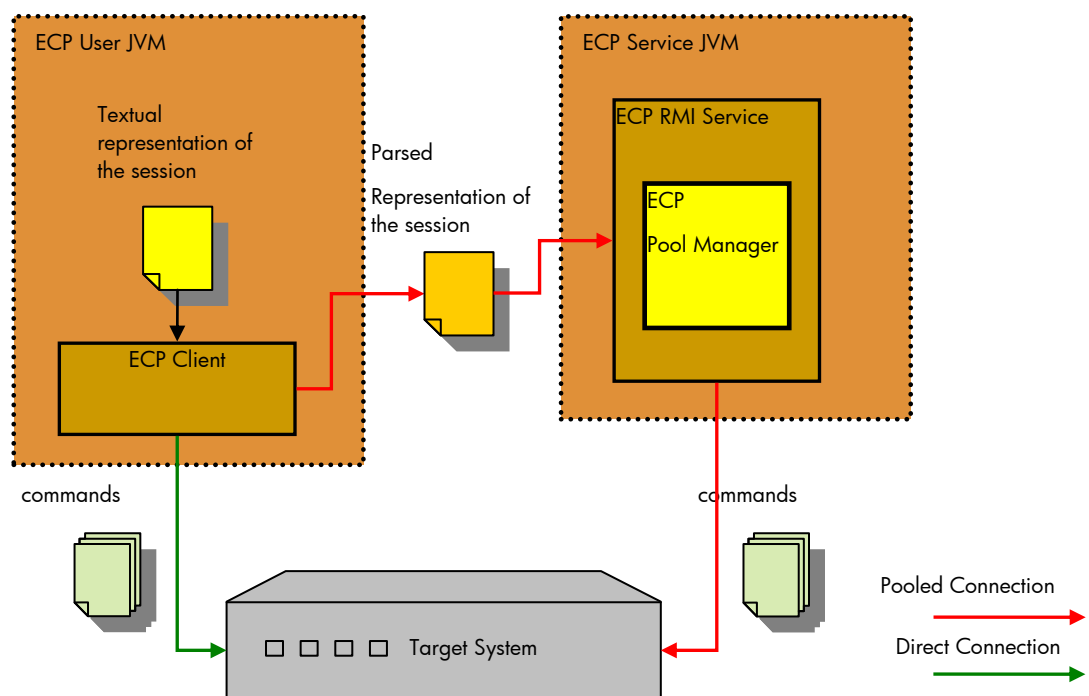
The function of the ECP Module, as part of the EP, is automating user interactive textual sessions, via TCP/IP connections to networked devices, such as routers, switches, proxies, etc...

The ECP Module receives a textual representation of the session, which states the commands to issue, their output and their meanings, and the control flow logic (such as the conditions under which a command must be issued or how many times must be issued).

The ECP Module is the module in EP which in the last instance directly connects to the EP managed devices, centralizing the EP management connections. This situation inside the EP framework is ideal to perform task such as load balancing, high availability and resources use optimization when referring to management connections. Toward this objective, the ECP Module implements a series of connections Pools, which provide the aforementioned functionalities, grouped in a Pool Manager.

The ECP Module is divided in two elements, the ECP Client and the ECP Service (an RMI service). The ECP Service receives the representations of the sessions and actually executes them, and contains the Pool Manager. The ECP Client acts mainly as a proxy, easing access to the ECP Service. It also allows the user to totally bypass the ECP RMI Service if needed, being the process transparent to the user. Bypassing the ECP Service is known as "Direct Connection" as opposed to "Pooled Connections" when using the ECP RMI Service. The use of either method is transparent to the user.

Such division allows easier scalability of the EP, while maintaining the ECP Module objectives of load balancing, high availability and resources use optimization.



1.3. ECP Module Entities and Concepts

1.3.1. Target System

In the context of the ECP, a "Target System" is the collection of resources accessible through a single direct TCP/IP Connection. Usually, a "Target System" will be a single router, switch or other similar device. However, more complex scenarios are possible if other devices are accessed from the connection end-point.

1.3.2. Operation

By "Operation" we refer to the collection of commands and logic needed to perform a certain process on the Target System. The purpose of the process may be a data inquiry, a configuration change or any other action needed on a Target System. An "Operation" should be atomic, that is, it should completely occur, or have no effects on the Target System. As a consequence, "Operations" should include the commands and logic needed to rollback the changes on the Target System if any. However, this policy is not enforced. Its use is left to the user's discretion.

1.3.3. Commands Template/Operation Template

A "Command Template" is a string which complies with a certain syntax through which an Operation is expressed, for the ECP Module to interpret and process it, usually with the purpose of automating a human interactive session on the Target System. The "Command Template" states the commands needed to perform the process (and usually to roll it back too), with specific information on every command, such as possible command outputs and their meaning (error, success) and the control flow which determines their execution order, among other things.

1.3.4. Operation Execution

An “Operation Execution” is the process through which Command Template is processed, resulting in commands inputted into the Target System.

1.3.5. Resource

In the context of the ECP, “Resource” is synonym of connection instance.

1.3.6. Pool

In the context of the ECP, a “Pool” is a set of established and authenticated connections (resources) to a single Target System that are kept ready to use. Each connection instance belongs to a single “Pool”. Connection instances life time is managed by the “Pool”. Pools are identified by name.

1.3.7. SubPool

A “SubPool” is a subset of the connections belonging to a Pool which are established with the Target System through the same interface, what generally implies though the same IP and Port (and user). The existence of the “SubPool” is only needed in the context of the ECP Configuration and Administration. In other contexts its use is transparent to the user. Each SubPool belongs to a single Pool. Every connection belongs to a single SubPool.

1.3.8. Equipment Driver

An “Equipment Driver” is a class whose instance encapsulates a single TCP/IP connection as a Pool Resource and is in charge of establishing, authenticating, verifying, and closing the underlying connection, when required by the Pool and as needed by the Target System. As some of this processes (especially authenticating, verifying and closing the connection) are dependent on the Target System type, usually a different “Equipment Driver” is needed for each Target System type, hence its name. It allows the developer and designer to easily add functionality to the ECP on per connection, per equipment, per equipment connection or even on connection event basis. Equipment Drivers must be provided by the ECP User.

The “Equipment Driver” is also in charge of executing every individual Commands Template command, that is: composing the Target System command, sending it to the Target System, reading the Target system answer, and interpreting it. Nevertheless, this functionality is provided by the ECP through inheritance.

For some tasks (such as establishing and closing the connection, or sending and reading data from it), the Equipment Driver will usually rely on a Protocol Driver to perform them as very often those task are not dependant on the Target System type, but on the network protocol to communicate with it. Entrusting this task on the Protocol Driver allows the programmer to reuse network protocol dependant functionality.

Typically, a different Equipment Driver is needed for each model of switch or router.

In the context of the ECP, the terms “connection”, “Resource”, and “Equipment driver”, are interchangeable.

1.3.9. Protocol Driver

A “Protocol Driver” is a class whose instance encapsulates a single TCP/IP connection, and is in charge of performing the most basic operations at low level, that is: establishing and closing the connection, sending and reading data from it, and encoding and decoding those data as needed by the Target System interface. Generally speaking, a Protocol Driver provides partial or total independence from the Application Layer of the OSI model. Entrusting this task on the Protocol Driver allows the programmer to reuse network protocol dependant functionality and the same Equipment Driver with different communication protocols.

The ECP provides Protocol Drivers for Telnet, SSH, and raw TCP network protocols.

1.3.10. ECP Instance

Since EP version 5.1-10, ECP has clustering support, meaning that there may be several ECP servers running at the same time in different machines with a shared database. Each one of those ECP servers is an ECP instance. Each ECP instance must be configured with a unique name to identify itself in the cluster.

1.4. General Architecture

On the highest level the ECP Module can be divided in two entities: the ECP Client and the ECP Service.

1.4.1. ECP Client

The ECP Client always is the entry point for the ECP user to the ECP Module, regardless of connection method or configuration.

The ECP Client is basically an ECP Service, without a Pool Manager. As such, it is able to execute Operations by itself and without the need of an ECP Service, opening and closing a new connection to the Target System for every Operation execution (Direct Connection), or delegating the execution of the Operation on the ECP Service (Pooled Connection). However, when using Direct Connections it can't profit on the aforementioned advantages of the RMI Service (load balancing, high availability and resources use optimization).

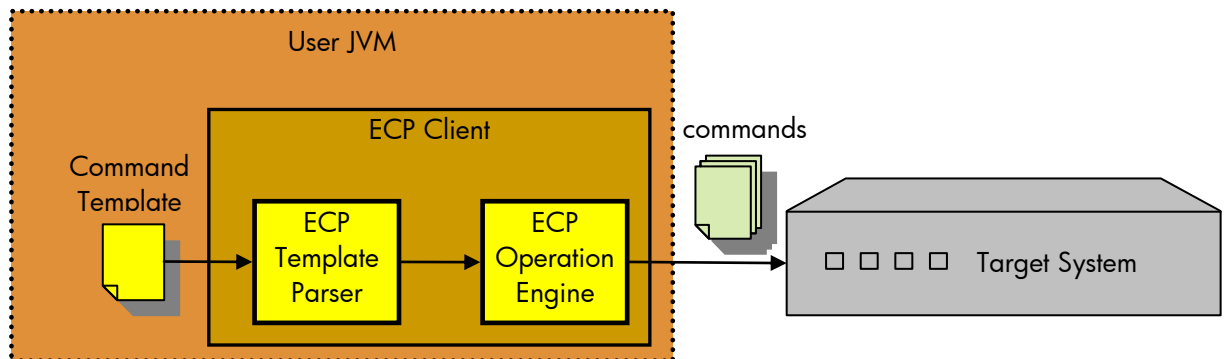
The ECP Client is constituted by two entities: The ECP Template Parser (`com.hp.spain.connection.TemplateParser`) and the ECP Operation Engine (`com.hp.spain.connection.CLICommands`).

The ECP Template Parser receives a Command Template (and some configuration) as input, returning an accordingly constructed ECP Operation Engine as a result.

The ECP Operation Engine receives connection configuration (and additional Operation commands if needed) as input, and when executed returns the session stdin and stdout or an exception if the Operation failed.

Depending on how the Template Parser and Operation Engine were configured, the real Operation execution will take place either locally (that is, in the client's Java Virtual Machine instance) or remotely (that is, in a different Java Virtual Machine instance)

The figure below represents a Direct Connection Operation execution. See the next section for an explanation of Pooled Connections.



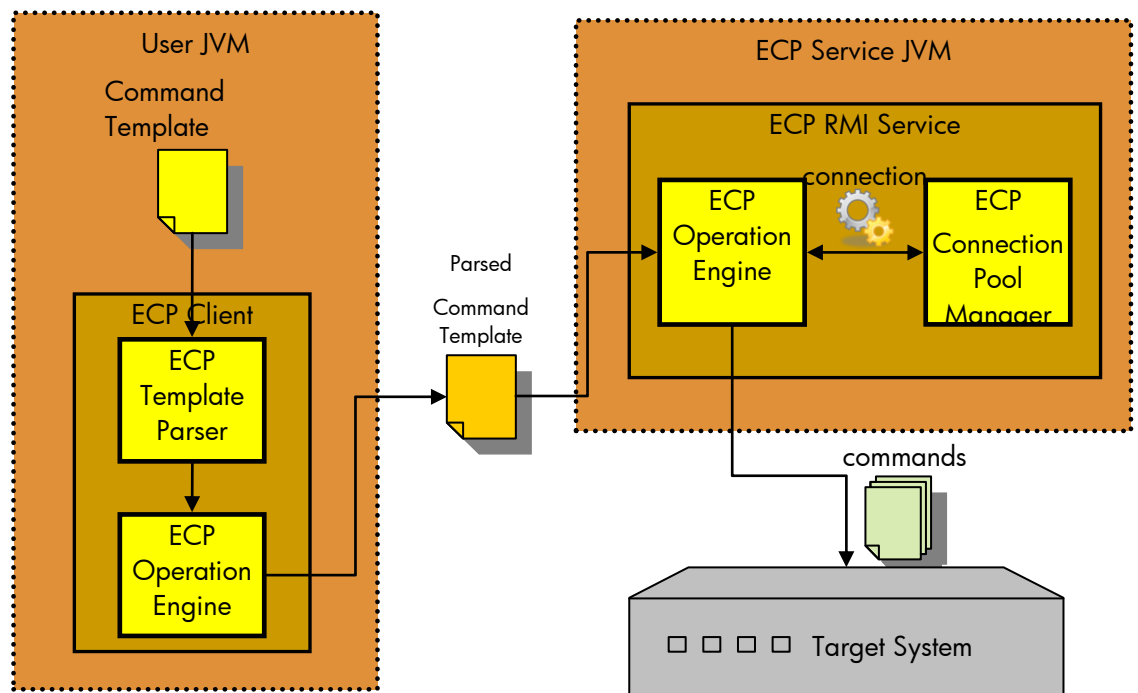
1.4.2. ECP Service

The ECP Service is basically an ECP Client which retrieves the connections to the Target System from a Pool Manager, instead of creating them.

If the ECP Client Operation Engine is configured to use Pooled Connections, on execution, instead of creating a connection, it will serialize the Parsed Command Template (contained by itself), and send it via RMI to the ECP Service.

On reaching the ECP RMI Service, The serialized Parsed Command Template will be used to instantiate an equivalent of the client's ECP Operation Engine. A connection from the Pool Manager will be assigned to this Operation Engine, which it will use to execute the Operation. The Operation will be executed as if from the client, but with a connection obtained from the Pool Manager instead. A different Operation Engine will be instantiated for each Operation, and multiple Operations may be executed concurrently.

The stdin and stdout or the failure of the Operation will be sent back to the caller Operation Engine (that is, the client's one).

Equipment Connection Pool

2. Functionality and Architecture

2.1. Connection and Pool Management

A single instance of the Pool Manager exists in the ECP Service. The Pool Manager contains a single Pool for each Target System (in a typical configuration).

Each Pool contains all the connections to a Target System, and is responsible of their life time and management. Additionally, it is responsible for:

- a) Connections reuse. The connections are kept alive, opened and authenticated, reusing the connections while possible.
- b) Identifying redundant interfaces on the Target System, and their connections, providing high availability.
- c) Queuing and prioritizing the Operation Engines' requests for connection to the Target System, providing load balance.
- d) Target System independence.
- e) Protocol independence.

2.1.1. Connection Reuse

Opening and maintaining a connection for each user is costly and wastes resources. On the contrary, pooling the connections enhances the performance of executing commands on a Target System. After a connection is created, it is placed in the Pool and reused over again while possible so that another connection does not have to be established and authenticated. The Pool creates (*initialize*) and destroys (*finalize*) new connections as needed, not exceeding the configured limits and politics. Connections are verified for consistency before being assigned to a client (*verify*). Additionally pooling the connections allows abstracting the client of the details of the connections management. Pooling the connections achieves reliable connections reuse.

2.1.2. High Availability

Every Pool may have one or more SubPools. Each SubPool represents a connection factory and container. Every SubPool comply the following rules:

- a) Each SubPool "owns" a different Target System interface. This means that all ECP connections to that Target System through that interface should be created and contained by the same SubPool instance.
- b) Connections from different SubPools should be equivalent, that is, executing an Operation through one or another SubPool should have the same effects on the Target System (provided the same initial Target System State).

Complying with this rules, allows the ECP to temporarily ignore a SubPool (interface) if it fails and becomes unusable (and another SubPool exists in the Pool), using the other SubPools (interfaces) instead. SubPooling the connections achieves high availability.

Since EP version 5.1-10, ECP has clustering support, taking high availability to the next level by allowing a given pool to be configured to run in several instances of the cluster. The administrator can designate an ordered list of ECP instances that should run the pool, as well as the number of instances of the pool that should exist at the same time, so the cluster functionality provides high availability and load balance features at the same time.

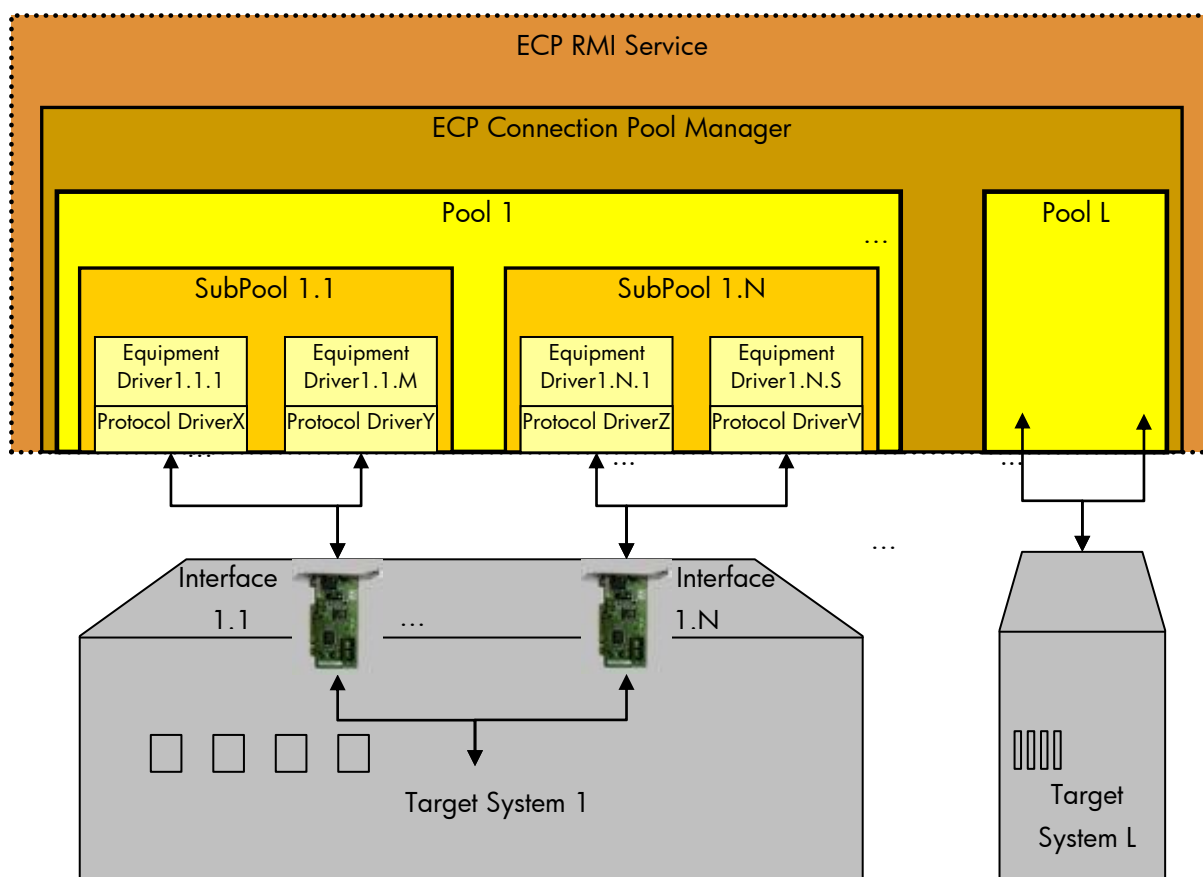
2.1.3. Target System Independence

The ECP needs to be able to connect, login, verify and disconnect the connections to the Target Systems as part of the Pooled connections management. As these processes are Target System specific, the ECP is unable to do so by itself. As a consequence, the ECP User must provide an Equipment Driver which performs those operations on behalf of the ECP. The Equipment Driver will wrap a connection, abstracting the ECP from the real tasks needed for those operations. Roughly speaking, the Equipment Driver scope is at a “per command” level.

2.1.4. Protocol Independence

Although Equipment Drivers perform Target System specific tasks, the underlying network protocol is usually standardized, and is not Target System dependant. For example, is very common for Target Systems to use SSH or Telnet protocols. To ease Equipment Driver development and allow protocol interchangeability, a Protocol Layer abstraction layer is implemented, called “Protocol Driver”. That layer will be responsible for establishing and closing the connection, sending and reading data from it, and encoding and decoding those data as needed by the Target System interface.

The ECP provides Protocol Drivers for Telnet, SSH, and raw TCP network protocols.



2.1.5. Load Balance

Every Pool, has a configurable number of “Request Queues”, where clients in need of a connection are kept waiting for their turn to acquire a connection. Queues can be prioritized, allowing critical Operations to remain as short as possible waiting for an available connection, and avoiding clients from becoming starved because of a high not critical Operations load. The priority of each request is established programmatically.

The frequency at which requests are dispatched and the number of available connections on each SubPool can be configured, allowing management of the load over the Target System and the ECP host. Request queuing and Pool size achieve load balance.

As stated in 2.1.2, new cluster functionality added to ECP in EP version 5.1-10 takes load balance support to a higher degree by allowing the distribution of pools across several ECP instances.

2.2. Pool and Connection types

Two types of Connection Pools are available, depending on how the pools are created.

2.2.1. Static vs Temporary Pools

ECP Module provides two different types of Pools: Static and Temporary.

Functionally, Temporary Pools are exactly the same as Static Pools, the only difference being that Temporary Pools will expire if unused for a configured amount of time, while static Pools will never expire.

Temporary Pools are useful when a Target System is going to be used for a short period of time and remain unused for long periods. Temporary Pools allow saving host resources in such situation.

When Pools are used, the Operation Execution is delegated on the ECP Service. See ECP Service.

2.2.2. Direct Connections (Not Pooled Connections)

When using Direct Connections, a connection is created for each executed Operation, being the connection private to the ECP Operation Engine instance used to issue the Operation. The Connection exists in the context of the ECP Operation Engine instance JVM. The Operation is executed in the JVM of the client. No ECP RMI Service is needed for this kind of Operation, although the Equipment Driver and Protocol Driver and their libraries will be needed. See ECP Client.

2.2.3. Dynamic Pools

The ECP Module allows the user to programmatically create Pools. Programmatically created Pools are referred “Dynamic Pools”. Dynamic Pools are usually temporary, although they can be static. As a consequence, “Dynamic Pools” aren’t created independently, but as part of the Operation Executions which uses them. This is due to the fact that a client can’t know whether the Dynamic Temporary Pool will still exist when the Operation Execution call is processed by the RMI ECP Service. For these reason, Operation Executions which use Dynamic Pools always carry the Dynamic Pool definition. On arrival to the ECP Service, the Dynamic Pool will be created if it does not exist. If it exists, the running Dynamic Pool instance will be used.

Since EP version 5.1-10, it is possible to store named dynamic pool configurations, so it is not necessary to provide all the required parameters for a dynamic pool at creation time. Dynamic pool configurations are currently supported only through the ECPCall node and require a new Micro-Workflow Manager module to be configured.

2.3. Commands Template

As “Commands Template” we understand a specially crafted String where, using a syntax specified by the ECP, the commands to Do, Undo, Commit and Rollback the Operation are established.

A Velocity Engine version 1.4 is provided with ECP, to ease the implementation of dynamic Commands Templates for the user. Through the method `TemplateParser#composeTemplate()`, a Velocity Commands Template can be easily merged with the data. See <http://velocity.apache.org/> for more details. See Commands Template Reference.

What follows is an example of a possible Commands Template:

```
[TEMPLATE:Do]

[TEMPLATE:Section 0]

show eth0 connections
  [TEMPLATE:EndStrPattern "admin#"]
  [TEMPLATE:Pattern "detination IP: (.*)"]
  [TEMPLATE:Array "destinationIPs"]

show eth1 connections
  [TEMPLATE:EndStrPattern "admin#"]
  [TEMPLATE:Pattern "detination IP: (.*)"]
  [TEMPLATE:Array "destinationIPs"]

[TEMPLATE:ForEach "var" In " destinationIPs"]
  ping %var% -n 1
  [TEMPLATE:EndStrPattern "admin#"]
[TEMPLATE:EndFor]

[TEMPLATE:Undo]
[TEMPLATE:Section 0]
```

The previous template executes queries connections through `eth0`, storing the destination IP in the array variable `destinationIPs`. The same process is repeated on `eth1`. After that, a ping is executed to all the obtained IPs. All commands are over when the prompt `admin#` is encountered. As the Template does not modify the Target System state, no Undo commands are needed.

2.4. Operation Execution

Operation Execution is the process through which the commands needed for the Operation to be done or undone are issued, appropriately handling the errors and rolling back the partial configuration change or committing the configuration changes.

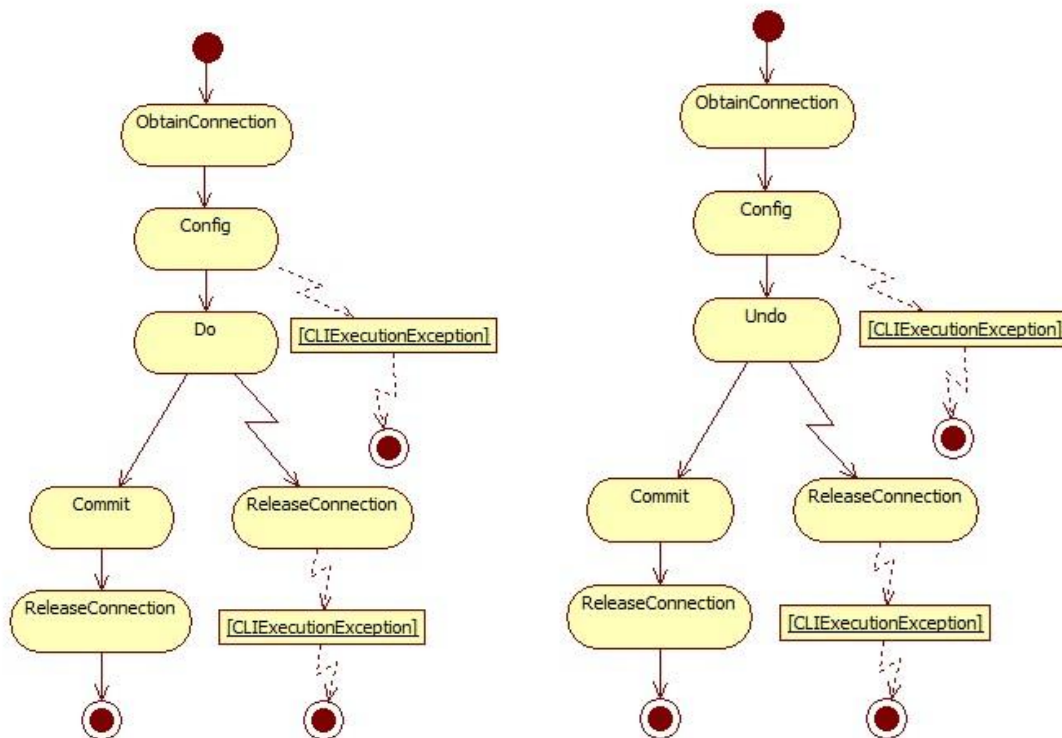
The client will provide a “Commands Template”, a specially crafted String where, using a syntax specified by the ECP, the commands to Do, Undo, Commit and Rollback the Operation are established. The Commands Template may contain conditional or looped execution of commands. Commands output

may be stored in variables and later used in conditions and commands. For a more detailed explanation see 2.3 Commands Template.

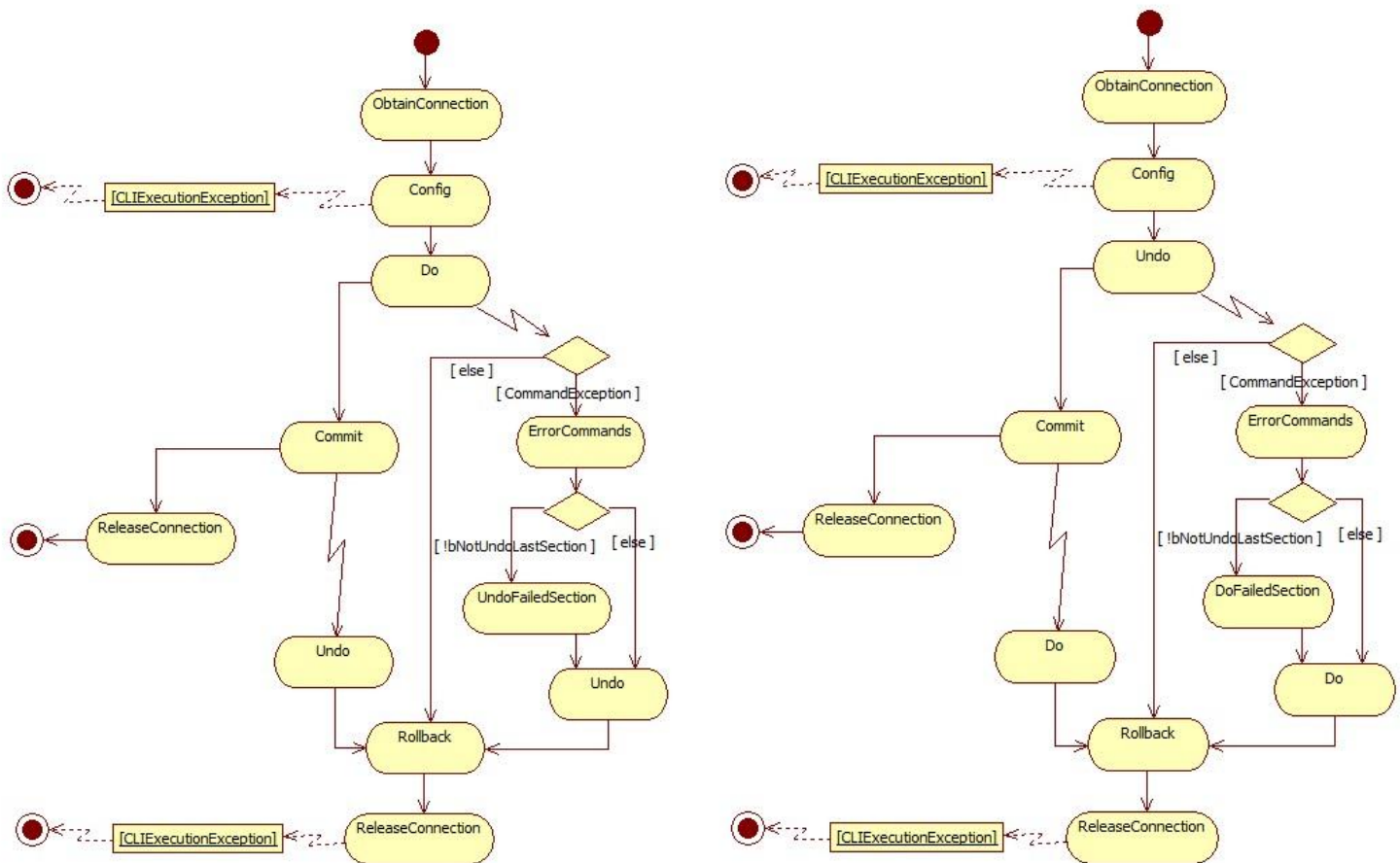
Each command in an Operation belongs to one of the following groups:

- a) Do: The commands collection to perform the configuration change.
- b) Undo: The commands collection to cancel the configuration change.
- c) Error: A set of commands to execute whenever a command output is identified as an unsuccessful command execution message.
- d) Commit: The commands to:
 - a. Make the configuration modifications effective/visible.
 - b. Save the configuration to a persistent media.
- e) Rollback: The commands to:
 - a. Restore the previous configuration from a persistent media.
 - b. Make the previous configuration effective/visible.

This is the recommended use for these groups, although other uses may be possible, always taking in to account the Do/Undo/Commit/Rollback logic. That logic is dependant on the call used to execute the Operation. Four methods are available: "Execute", "ExecuteActivation", "Revert" and "InverseActivation". See the following diagrams for more detail:



"Execute" and "Revert" activity diagrams.



"ExecuteActivation" and "InverseActivation" activity diagrams.

2.5. Real-time Monitoring

From EP version 2.3 onwards the ECP is able to provide real time information of its execution through JMS. Currently, ECP includes Active MQ 4.1.1 which fully implements JMS 1.1. If JMS monitoring is enabled, ECP may start its own embedded JMS service (by default) or connect to a remote one. Active MQ includes many features, like persistent, transactional and XA messaging; message groups, virtual destinations, wildcards and composite destinations; pluggable transport protocols as TCP, SSL, UDP, in-VM (embedded); clustering; bridging to other JMS providers; JMX administration, etc...

3. First Steps

3.1. Equipment Driver Development

3.1.1. Equipment Driver Development Introduction

As previously said (see Equipment Driver) an “Equipment Driver” is a class which encapsulates a single connection as an ECP Resource (a pooled connection). As such, an Equipment Driver must implement the functionality expected by the ECP, that is:

- a) The capability of executing commands.
- b) The ability of behaving as an ECP Connection.

The main part of that functionality is already implemented and inherited from the driver parent classes `EquipmentDriver` and `ConnectionResource`, simplifying the driver development.

The capability of executing commands is fully provided by the `EquipmentDriver` class, very rarely requiring additional implementation or overriding in the driver.

The ability of behaving as an ECP Connection is partially provided by the `ConnectionResource` class, and as a consequence, additional implementation and overriding will be needed in the driver.

In addition, an instance of another class, the `ProtocolDriver`, will provide protocol independence, allowing the use of the same Equipment Driver with varying communications protocols (telnet, SSH, etc...) to the destination equipment.

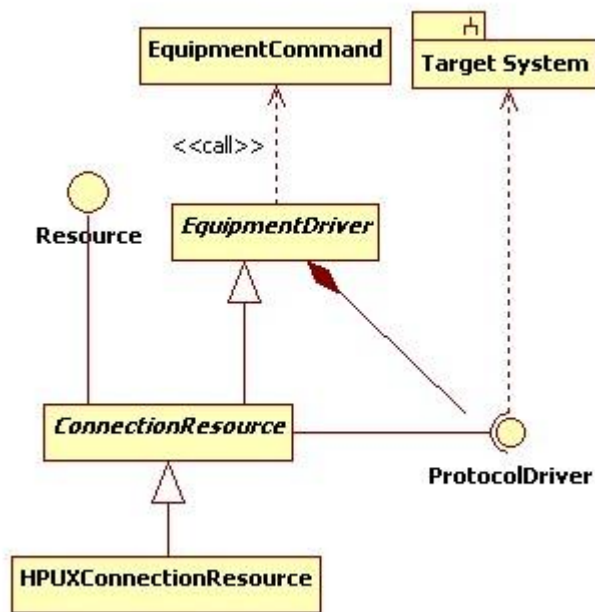
While developing the driver, the programmer must be careful not to choose libraries versions which differ of the versions present in `$ECP_LIB` (if Pooled Connections are used) and/or the versions present in the `$ACTIVATOR_THIRD_PARTY/lib` (if Direct Connections are used). The Equipment Driver classes and its dependencies may have to be deployed in one or both of those paths and the driver classes will be loaded using the same `ClassLoader` as the rest of libraries there. See Equipment Driver Deployment for further details.

See Available Equipment Drivers for a list of some Equipment Drivers already implemented. Notice that that list includes only the precise versions of the Target Systems against which a certain driver has been or is being used in a production environment. These drivers might be compatible as is with some other Target Systems or versions, or might be easily adapted to them.

3.1.1.1. Equipment Driver Classes

Every Equipment Driver must inherit from `com.hp.spain.connection.ConnectionResource`. The following diagram shows the typical class diagram of an Equipment Driver example (`HPUXConnectionResource`).

Equipment Connection Pool



i. *com.hp.spain.connection.ProtocolDriver Class*

A "Protocol Driver" is a class whose instance encapsulates a single TCP/IP connection, and is in charge of performing the most basic operations at low level, that is: establishing and closing the connection, sending and reading data from it, and encoding and decoding those data as needed by the Target System interface. Generally speaking, a Protocol Driver provides partial or total independence from the Application Layer of the OSI model. Entrusting this task on the Protocol Driver allows the programmer to reuse network protocol dependant functionality and the same Equipment Driver with different communication protocols.

The ECP provides Protocol Drivers for Telnet, SSH, and raw TCP network protocols.

See `com.hp.spain.connection.ProtocolDriver` Class for further information.

ii. *com.hp.spain.connection.EquipmentCommand Class*

This class encapsulates the information needed to execute a command on the Target System, that is, to construct the string to be sent, send it and read its output, interpret it, and extract information from it.

iii. *com.hp.spain.connection.EquipmentDriver Class*

This class contains the functionality needed to execute a command on the Target System represented as an `EquipmentCommand`, that is, construct the string to be sent, send it, and process its output. It also contains some basic connection operations, such as establishing a connection, closing a connection, and authenticating.

iv. *com.hp.spain.connection.Resource Class*

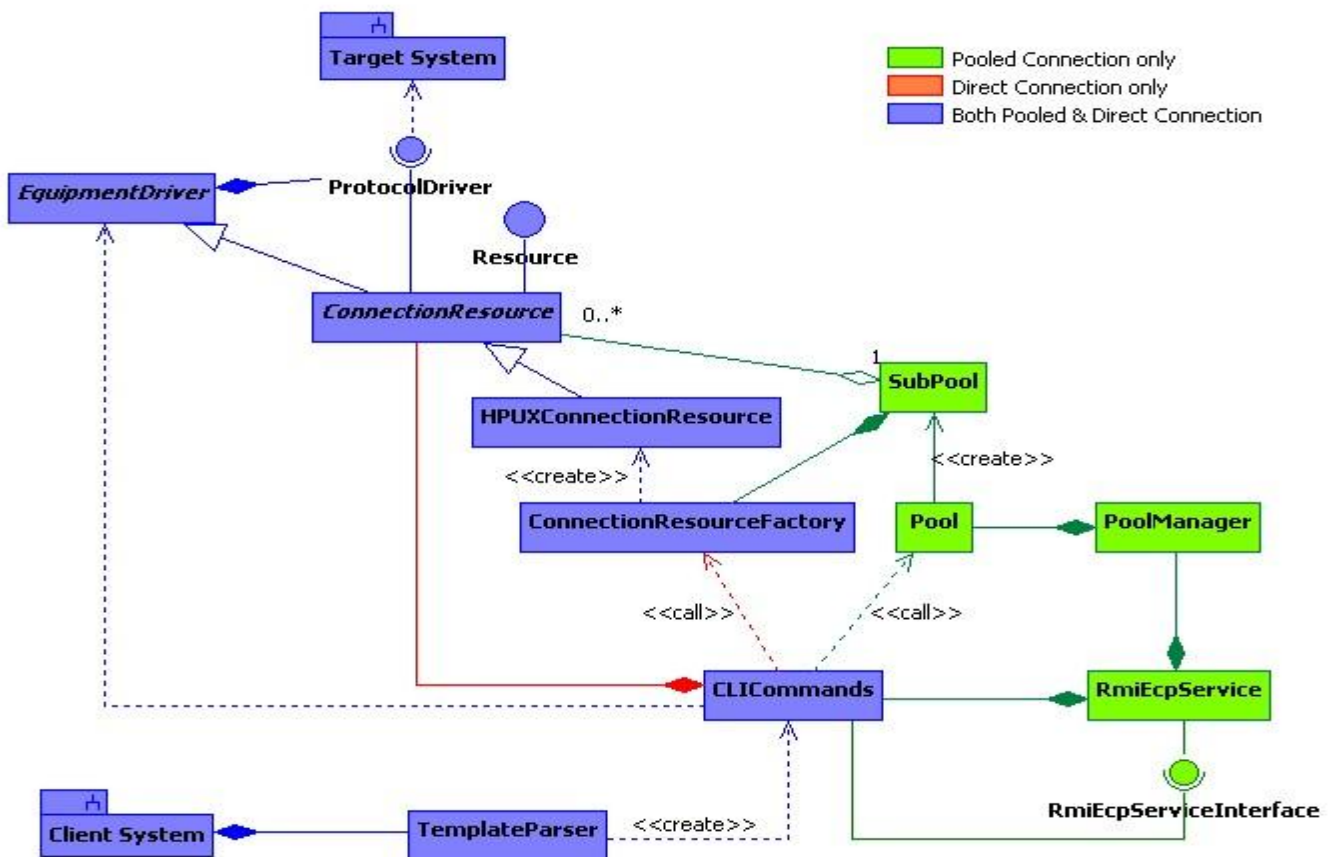
This interface represents a basic pooled object. Defines the operations needed to manage an object belonging to a pool.

v. *com.hp.spain.connection.ConnectionResource Class*

This class implements an EquipmentDriver as a pooled object. It implements the functionality defined by *Resource* using the operations provided by *EquipmentDriver*, that is, executing commands and basic connection operations.

3.1.1.2. Equipment Driver inside the ECP

The following diagram shows an example Equipment Driver (*HPUXConnectionResource*) and its relation with the ECP.



3.1.2. Equipment Driver Generic

This driver is a usefull class to avoid the implementation of the different driver's states. Then when the developer has to make a new driver just focus on the equipments requirements.

The driver has to extend the class *com.hp.spain.connection.EquipmentDriverGeneric* and overwrite next methods. Some of them are optional.

- `void verifyLoggedIn():` optional. Makes the verification when the driver status is loggedin
- `void verifyConnected():` optional. Makes the verification when the driver status is connected
- `void verifyConfigMode():` optional. Makes the verification when the driver status is configMode

Equipment Connection Pool

- void verifyUnknownMode(): optional. Makes the verification when the driver status is unknown
- void enterConfigMode(): optional. Execute the commands required to config the connection
- void exitConfigMode(): optional. Execute the commands required to unconfig the connection
- void logout(): optional. Executes the commands to logout the connection
- void initializeSpecificParameters(String specificParameters): optional.
- void waitForLoginUserPrompt(): usually required for protocol without authentication support. Synchronize the login prompt.
- void waitForLoginPwdPrompt(): usually required for protocol without authentication support. Synchronize the login password.
- void waitForInitialCommandPrompt(): usually required. Synchronize the initial prompt.

3.1.3. Equipment Driver Deployment

The Equipment Driver may have to be deployed in two different paths, depending on the type of connection used.

For Pooled Connections, the Equipment Driver jar and its dependencies must be placed inside the `$ECP_LIB` directory, and the ECP Service restarted. The ECP Service should be restarted whenever that directory contents are modified for the ECP to incorporate the changes. For the Equipment Driver to be instantiated a Static or Dynamic Pool which uses that Equipment Driver must be created and depending on the ECP configuration, even a Commands Template executed against it.

For Direct Connections, the Equipment Driver jar and its dependencies must be placed inside the `$ACTIVATOR_THIRD_PARTY/lib` directory. The Micro Workflow Manager and the Resource Manager must be restarted for the changes to take effect. The Micro Workflow Manager and the Resource Manager should be restarted whenever that directory contents are modified. A Commands Template must be executed for the Equipment Driver to be instantiated.

3.1.4. Available Equipment Drivers

The following Equipment Drivers have been already developed and are available:

| Equipment Driver | Tested On | | | | Use Case | | |
|---------------------|------------------|---|---------------|-----------------------|---|--|----------|
| | Manufacturer | Model | Sw/Fw Version | Type | Context | Operations | Protocol |
| cisco-ovsa-plugin | Cisco | Catalyst 2960 Series Switch. Unknown Models | ? | Ethernet Switch | Level 2 Network | VLAN Configuration, ACL's, DHCP | Telnet |
| | | Catalyst 4503 Switch | ? | | | | |
| oxe-ovsa-plugin | Alcatel-Lucent | OmniPCX Enterprise | 7.1 8.1 | Communications Server | VoIP Telephone Exchange | Channel Administration | Telnet |
| juniper-ovsa-driver | Juniper Networks | M40e | ? | Router | Provider Router IP/MPLS Network Access Router | Configuration Configuration diagnosis | SSH |
| teldat-ovsa-driver | Teldat | Atlas 250 | ? | Router | Client Router | Configuration diagnosis | Telnet |

| | | | | | | | |
|------------------------|------------|---------------------------------|---|-----------------|---------------|-------------------------|--------|
| cisco-ovsa-driver | Cisco | 2801 Integrated Services Router | ? | Router | Client Router | Configuration diagnosis | Telnet |
| | | 2621XM Multiservice Router | ? | | | | |
| catalyst-ovsa-driver | Cisco | Catalyst 3560-24TS | ? | Ethernet Switch | Client Router | Configuration diagnosis | Telnet |
| | | Catalyst 3560-48TS | ? | | | | |
| | | Catalyst 3550-24-EMI | ? | | | | |
| | | Catalyst 3550-12G | ? | | | | |
| riverstone-ovsa-driver | Riverstone | RS1100 | ? | Router | Client Router | Configuration diagnosis | Telnet |
| | | RS 3100 | ? | | | | |

Notice that this list includes only the precise versions of the Target Systems against which a certain driver has been or is being used in a production environment. These drivers might be compatible as is with some other Target Systems or versions, or might be easily adapted to them.

3.1.5. Generic Template Equipment Driver

This driver is able to connect to any type of equipments using some variables or templates. The most important feature of this driver is the capability to connect any equipment and not require any java development.

This equipment driver is configured using the class `com.hp.spain.connection.TemplateDriver`. We can configure this driver adding into the `DriverSpecificParameters` the extra variables on properties format or referring to the Common Configuration.

The next templates can be configured into the database or into a file, finding first in database.

- `LOGIN_TEMPLATE`: template to make the login (note: this template has sense in protocol driver that doesn't make the authentication)
- `LOGOUT_TEMPLATE`: the logout template, typically the exit command
- `ENTER_CONFIG_MODE_TEMPLATE`: the template to configure all the sessions attributes required.
- `EXIT_CONFIG_MODE_TEMPLATE` -> the template to unconfigure
- `VERIFY_TEMPLATE`: template to verify if the connection is ok.

These templates will receive the parameters configured into the `DriverSpecificParameters` and the next parameters configured into the subpool:

- `USER`: user name
- `PASSWORD`: user password
- `PASSWORD_ENABLE`: password enable
- `HOST`: ip host value

Also, the next variables can be defined to make easier the templates:

- `LOGIN_USER_PROMPT`: synchronize the driver with the login prompt.
- `LOGIN_PWD_PROMPT`: synchronize the driver with the password prompt.
- `INITIAL_PROMPT`: synchronize the driver with the initial prompt.

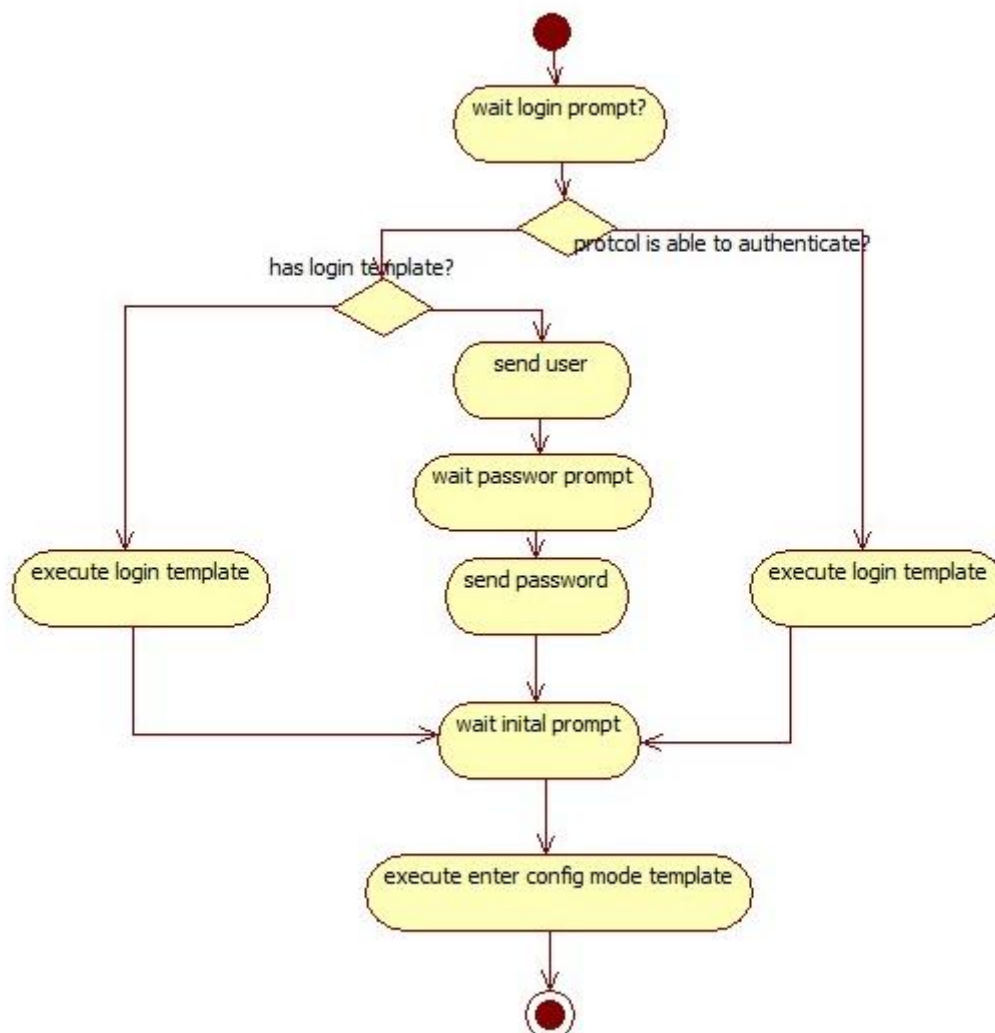
Equipment Connection Pool

Also, this driver has the capability to add error patterns, failure patterns, non error patterns and error message to all the commands that are executed into a command template. In case, it's required to add these patterns to the connections templates (LOGIN_TEMPLATE, ENTER_CONFIG_MODE_TEMPLATE, ...) the variable ADD_PATTERNS_CONNECTION_TEMPLATES has to be setted to true.

The only requirement to set these patterns is define variables with next prefix:

- ENDSTRING_PATTERN
- ERROR_PATTERN
- FAILURE_PATTERN
- NONERROR_PATTERN
- ERROR_MESSAGE: in this case only can be defined one and the variable is required to have this name.

3.1.5.1. Connection



When the driver starts the connection the first step is to check if the LOGIN_USER_PROMPT is configured. In that case, synchronize this prompt. After that, the LOGIN_TEMPLATE is executed if it's configured. If not

and the protocol driver doesn't support authentication, send the user, synchronize the password prompt (LOGIN_PWD_PROMPT) and send the password.

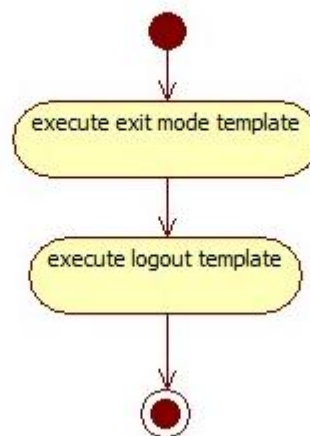
In this moment, the driver is authenticated and in case the INITIAL_PROMPT is configured the driver synchronizes the initial prompt.

Usually, when the protocol supports the authentication (for example, ssh) it's only necessary to configure the INITIAL_PROMPT and not the LOGIN_TEMPLATE and neither LOGIN_USER_PROMPT.

After synchronize the INITIAL_PROMPT the driver execute the ENTER_CONFIG_MODE_TEMPLATE and finally executes the VERIFY_TEMPLATE.

In this moment, the driver is connected and ready to be used.

3.1.5.2. Disconnect



First, the driver execute the template EXIT_CONFIG_MODE_TEMPLATE and after that the LOGOUT_TEMPLATE.

3.1.5.3. Examples of DriverSpecificParameters

Telnet easiest configuration:

```

LOGOUT_TEMPLATE=logout.vm
LOGIN_USER_PROMPT=.*login\:
LOGIN_PWD_PROMPT=.*password\:
INITIAL_PROMPT=C\:.*\>
  
```

Ssh easiest configuration:

```

INITIAL_PROMPT=#
  
```

Telnet using login template and patterns:

```

LOGOUT_TEMPLATE=logout.vm
LOGIN_USER_PROMPT=.*login\:
LOGIN_TEMPLATE=login.vm
VERIFY_TEMPLATE=/opt/HP/OV/ECP/templates/verify.vm
ENDSTRING_PATTERN= C\:.*\>
ERROR_PATTERN_1=ERROR [0-9]+ .*
  
```

```

ERROR_PATTERN_2=[0-9][0-9] ERROR .*
FAILURE_PATTERN_1=FAILURE [0-9]+ .*
FAILURE_PATTERN2=[0-9][0-9] FAILURE .*
ERROR_MESSAGE=error message
NONERROR_PATTERN_1=Warning .*

```

Ssh using enter mode config template and patterns:

```

LOGOUT_TEMPLATE=logout_ssh.vm
LOGIN_TEMPLATE=login.vm
ENTER_CONFIG_MODE_TEMPLATE=enterConfigMode.vm
EXIT_CONFIG_MODE_TEMPLATE=exitConfigMode.vm
VERIFY_TEMPLATE=/opt/HP/OV/ECP/templates/verify_ssh.vm
INITIAL_PROMPT=\\[forge\\]\\$
ENDSTRING_PATTERN=\\[forge\\]\\$
ERROR_PATTERN_1=ERROR [0-9]+ .*
ERROR_PATTERN_2=[0-9][0-9] ERROR .*
FAILURE_PATTERN_1=FAILURE [0-9]+ .*
FAILURE_PATTERN2=[0-9][0-9] FAILURE .*
ERROR_MESSAGE=error message
NONERROR_PATTERN_1=Warning .*

```

3.2. ECP Service Process

3.2.1. Starting ECP Service

To start the ECP Service, use the following:

Windows:

```
$ECP_BIN\StartServer.bat
```

On Unix:

```
$ECP_BIN\StartServer.sh
```

3.2.2. Stopping ECP Service

To stop the ECP Service, use the following:

Windows:

```
$ECP_BIN\StopServer.bat
```

On Unix

```
$ECP_BIN\StopServer.sh
```

3.2.3. Restarting ECP Service

Just stop and start the ECP Service.

3.2.4. Checking ECP Service

To check the ECP Service, use the following:

Windows:

```
$ECP_BIN\showStatus.bat
```

On Unix:

```
$ECP_BIN\showStatus.sh
```

3.3. Use Examples

What follows is a series of examples of ECP Clients. In those examples, the following class, simulating a client configuration, is used. You will probably have some other particular way of obtaining the configuration. Notice that, depending on the connection type, not all of the configuration parameters are needed:

```
package com.hp.spain.connection.pool.examples;

public class ExamplesConfiguration {

    //Commands Template
    private static String Template =
        "[TEMPLATE:Do]\n" +
        "[TEMPLATE:Section 0]\n" +
        "help\n" +
        "          [TEMPLATE:EndStrPattern \"nina.*\"]\n" +
        "          [TEMPLATE:Error \"%CLI-E-NOFACINST, no facility instance allowed\"]\n" +
        "[TEMPLATE:Undo]\n" +
        "[TEMPLATE:Section 0]\n";

    //Target system data
    private static String Hostname="172.16.2.111";
    private static int Port=23;
    private static String Login="admin";
    private static String Password="pass4hpsa";
    private static String PasswordEnable="pass4hpsa";

    //Drivers data
    private static String Protocol="telnet";
    private static String ConnectionResourceClassName =
        "com.hp.spain.connection.RiverstoneRSConnectionResource";
    private static String AdditionalData="Other needed values";

    //Pool data
    private static String PoolName="examplePool";
    private static int MaxCon=3;
    private static int MinCon=1;
    private static boolean InitOnCreate=true;
    private static int OverMinimumConnTimeout=30000;
    private static int ReservedConnTimeout=60000;
    private static int PoolTimeout=600000;

    //ECP Instance data
    private static String ECPHost="127.0.0.1";
    private static String ECPPort="1200";
    private static int QueueID=1;

    public static String getAdditionalData() {
        return AdditionalData;
    }
    public static String getConnectionResourceClassName() {
        return ConnectionResourceClassName;
    }
    public static String getECPHost() {
        return ECPHost;
    }
    public static String getECPPort() {
        return ECPPort;
    }
    public static String getHostname() {
        return Hostname;
    }
    public static boolean isInitOnCreate() {
        return InitOnCreate;
    }
    public static String getLogin() {
```

```

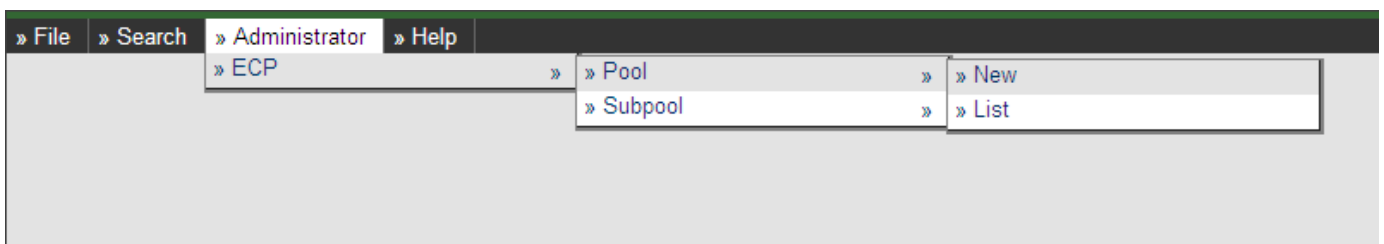
        return Login;
    }
    public static int getMaxCon() {
        return MaxCon;
    }
    public static int getMinCon() {
        return MinCon;
    }
    public static int getOverMinimunConnTimeout() {
        return OverMinimunConnTimeout;
    }
    public static String getPassword() {
        return Password;
    }
    public static String getPasswordEnable() {
        return PasswordEnable;
    }
    public static String getPoolName() {
        return PoolName;
    }
    public static int getPoolTimeout() {
        return PoolTimeout;
    }
    public static int getPort() {
        return Port;
    }
    public static String getProtocol() {
        return Protocol;
    }
    public static int getReservedConnTimeout() {
        return ReservedConnTimeout;
    }
    public static String getTemplate() {
        return Template;
    }
    public static int getQueueID() {
        return QueueID;
    }
}

```

3.3.1. Creating and Using an Static Pool

In the source examples a Pool called “examplePool” will be used. What follows is a quick guide to create a Pool. Refer to the document “OVSA EP for Service Providers - ECP Administration GUI - User Reference” for details on how to administer Pools and SubPools using the ECP GUI.

First, create the Pool (menu “Administrator->Pool->New”)



Fill in the formulary that will appear.

- Name: “examplePool”. The Pool ID will be used from code to reference to the pool.
- Log File: “examplePool.log”. Name of the file where the Pool activity will be logged.
- Log Level: info
- Maximum Pool Life Time from...: 10000
- Weights: 1,2,3,4,5

Pool Creation

»Name: examplePool »Log file: examplePool.log

»Log level: Info »Requests timeout (ms): 30000

»Maximum Pool Life Time from its last use (ms): 0

Priority weighed queues

| Priorities: | »1 | »2 | »3 | »4 | »5 |
|-------------|----|----|----|----|----|
| Weights: | 1 | 2 | 3 | 4 | 5 |

Save the Pool (menu "Pool->Save")

Pool Creation

»Name: examplePool »Log file: examplePool.log

»Log level: Info »Requests timeout (ms): 30000

»Maximum Pool Life Time from its last use (ms): 0

Priority weighed queues

| Priorities: | »1 | »2 | »3 | »4 | »5 |
|-------------|----|----|----|----|----|
| Weights: | 1 | 2 | 3 | 4 | 5 |

After creating the Pool, create a SubPool (menu "Administrator->SubPool->New")

» File » Search » Administrator » ECP » Pool » Subpool » New

» List

Fill in the formulary that will appear.

- Pool Name: "examplePool". The Pool to which this SubPool belongs.
- Min. Sessions: 10
- Max Sessions: 100
- Init Sessions: 1
- Temporary Sessions life Time=1000000
- Max. Sessions use time= 100000000

The rest of the values are dependent on the Target System. These values are given as an example

- Equipment Connection Resource Class: Class of the Equipment driver. For example: "com.hp.spain.connection.RiverstoneRSConnectionResource"
- IP: Target System IP. For example: 172.16.2.111
- Protocol: Protocol Driver to use. For example: telnet
- Port: Port to connect through to the Target System. For example: 23
- User: User Name to log into the Target System. For example: admin
- Password: Password to log into the Target System.

Equipment Connection Pool

Subpool

SubPool Creation

| | | | |
|--|----------------------------------|---------------------------------------|---------|
| » Pool name:* | examplePool | » Protocol:* | telnet |
| » Equipment connection resources class:* | n.RiverstoneRSConnectionResource | » Port:* | 23 |
| » IP:* | 16.38.0.138 | » Password :* | ***** |
| » User:* | admin | » Max. Sessions:* | 3 |
| » Password of enable: | ***** | » Temporary sessions life time (ms):* | 30000 |
| » Min. Sessions:* | 1 | » Max. Sessions use time (ms):* | 1800000 |
| » Init. Sessions:* | 1 | | |
| » Driver specific parameters:* | | | |

Save the SubPool (menu "SubPool->Save")

Subpool

Save

SubPool Creation

| | | | |
|--|----------------------------------|---------------------------------------|---------|
| » Pool name:* | examplePool | » Protocol:* | telnet |
| » Equipment connection resources class:* | n.RiverstoneRSConnectionResource | » Port:* | 23 |
| » IP:* | 16.38.0.138 | » Password :* | ***** |
| » User:* | admin | » Max. Sessions:* | 3 |
| » Password of enable: | ***** | » Temporary sessions life time (ms):* | 30000 |
| » Min. Sessions:* | 1 | » Max. Sessions use time (ms):* | 1800000 |
| » Init. Sessions:* | 1 | | |
| » Driver specific parameters:* | | | |

What follows is an example of static Pool use:

```
package com.hp.spain.connection.pool.examples;

import java.util.HashMap;

import com.hp.spain.connection.CLICommands;
import com.hp.spain.connection.CLIExecutionException;
import com.hp.spain.connection.TemplateParser;
import com.hp.spain.connection.TemplateParserException;

public class StaticPoolConnExample {

    public static void main (String[] args)
    throws CLIExecutionException, TemplateParserException {
        HashMap oRet=null; //the Operation execution result
        TemplateParser parser; //the ECP Template Parser instance
        CLICommands cliCommands; //the ECP Operation Engine instance

        //ECP Template Parser instantiation and configuration
        parser=new TemplateParser();

        //ECP Operation Engine instantiation and configuration
        cliCommands = parser.parseTemplate(ExamplesConfiguration.getTemplate());

        //ECP instance
        cliCommands.setRMIHostName(ExamplesConfiguration.getECPHost());
        cliCommands.setRMIPort(ExamplesConfiguration.getECPPort());

        //Operation execution
        oRet=cliCommands.execute(
            ExamplesConfiguration.getPoolName(),
            ExamplesConfiguration.getQueueID());
        //Other possible executions would have been:
        //oRet=cliCommands.executeActivation(
        //    ExamplesConfiguration.getPoolName(),
        //    ExamplesConfiguration.getQueueID());
        //oRet=cliCommands.revert(
        //    ExamplesConfiguration.getPoolName(),
```

```
//      ExamplesConfiguration.getQueueID();
//oRet=cliCommands.inverseActivation(
//      ExamplesConfiguration.getPoolName(),
//      ExamplesConfiguration.getQueueID());

//Execution Output
System.out.println("RESULT HASHMAP:");
System.out.println(oRet);
System.out.println("COMMANDS SENT:");
System.out.println(cliCommands.getCommandsSent());
System.out.println("STDOUT:");
System.out.println(cliCommands.getStdOut());
}
}
```

To execute the example, run the following command, where <classpath> should contain all the libraries contained in the directory \$ECP_LIB plus the path where these classes have been compiled. The java version must be 1.4.2.

```
java -classpath <classpath> com.hp.spain.connection.pool.examples.StaticPoolConnExample
```

3.3.2. Creating and Using a Dynamic Pool

It is possible to create a Pool programmatically, indicating its properties as part of an Operation Execution. Programmatically created pools are called “Dynamic Pools” and are usually temporary. See Dynamic Pools for a more detailed explanation.

```
package com.hp.spain.connection.pool.examples;

import java.util.HashMap;

import com.hp.spain.connection.CLICommands;
import com.hp.spain.connection.CLIExecutionException;
import com.hp.spain.connection.TemplateParser;
import com.hp.spain.connection.TemplateParserException;
import com.hp.spain.connection.pool.DynamicEcpProperties;

public class DynPoolConnExample {

    public static void main (String[] args)
    throws CLIExecutionException, TemplateParserException {
        HashMap oRet=null; //the Operation execution result
        TemplateParser parser; //the ECP Template Parser instance
        CLICommands cliCommands; //the ECP Operation Engine instance
        DynamicEcpProperties oDynProps;

        //ECP Template Parser instantiation and configuration
        parser=new TemplateParser();
        //Target System Data
        parser.setHostname(ExamplesConfiguration.getHostname());
        parser.setPort(ExamplesConfiguration.getPort());
        parser.setLogin(ExamplesConfiguration.getLogin());
        parser.setPassword(ExamplesConfiguration.getPassword());
        parser.setPasswordEnable(ExamplesConfiguration.getPasswordEnable());

        //ECP Operation Engine instantiation and configuration
        cliCommands = parser.parseTemplate(ExamplesConfiguration.getTemplate());
        //Equipment and Protocol Drivers
        cliCommands.setProtocol(ExamplesConfiguration.getProtocol());
        cliCommands.setConnectionResourceClassName(
            ExamplesConfiguration.getConnectionResourceClassName());
        //Pooling Data
        //Pool name is ptional. By default dynamic pool names are autogenerated.

        cliCommands.setDynamicPoolName(ExamplesConfiguration.getPoolName());
        oDynProps= cliCommands.getDynamicEcpProperties();
    }
}
```

```

oDynProps.setPoolConfiguration(
    //maximum number of connections to be contained in the pool
    ExamplesConfiguration.getMaxCon(),
    //minimum number of connections to be contained in the pool
    ExamplesConfiguration.getMinCon(),
    //initialize on instantiation, instead of on first use
    ExamplesConfiguration.isInitOnCreate(),
    //Not used timeout of connections over the minimum (ms)
    ExamplesConfiguration.getOverMinimumConnTimeout(),
    //maximum time a connection may be in use by an Operation (ms)
    ExamplesConfiguration.getReservedConnTimeout(),
    //Not used timeout for the pool
    ExamplesConfiguration.getPoolTimeout()
);
//ECP instance
cliCommands.setRMISHostName(ExamplesConfiguration.getECPHost());
cliCommands.setRMIPort(ExamplesConfiguration.getECPPort());

//Equipment Driver additional initialization parameters
oDynProps.setSpecificParameters(ExamplesConfiguration.getAdditionalData());

//Operation execution
oRet=cliCommands.execute(oDynProps, ExamplesConfiguration.getQueueID());

//Other possible executions would have been:
//oRet=cliCommands.executeActivation(oDynProps, ExamplesConfiguration.getQueueID());
//oRet=cliCommands.revert(oDynProps, ExamplesConfiguration.getQueueID());
//oRet=cliCommands.inverseActivation(oDynProps, ExamplesConfiguration.getQueueID());

//Execution Output
System.out.println("RESULT HASHMAP:");
System.out.println(oRet);
System.out.println("COMMANDS SENT:");
System.out.println(cliCommands.getCommandsSent());
System.out.println("STDOUT:");
System.out.println(cliCommands.getStdOut());
}
}

```

To execute the example, run the following command, where <classpath> should contain all the libraries contained in the directory \$ECP_LIB plus the path where these classes have been compiled. The java version must be 1.4.2.

```
java -classpath <classpath> com.hp.spain.connection.pool.examples.DynPoolConnExample
```

3.3.3. Using Direct Connections

It is possible to bypass the ECP Service when executing a Commands Template. See Direct Connections (Not Pooled Connections) for a detailed explanation.

```

package com.hp.spain.connection.pool.examples;

import java.util.HashMap;

import com.hp.spain.connection.CLICommands;
import com.hp.spain.connection.CLIExecutionException;
import com.hp.spain.connection.ConnectionResource;
import com.hp.spain.connection.TemplateParser;
import com.hp.spain.connection.TemplateParserException;

public class DirectConnExample {

    public static void main (String[] args)
        throws CLIExecutionException, TemplateParserException {
        HashMap oRet=null; //the Operation execution result
        TemplateParser parser; //the ECP Template Parser instance
        CLICommands cliCommands; //the ECP Operation Engine instance

        //ECP Template Parser instantiation and configuration
        parser=new TemplateParser();
    }
}

```

```
//Target System Data
parser.setHostname(ExamplesConfiguration.getHostname());
parser.setPort(ExamplesConfiguration.getPort());
parser.setLogin(ExamplesConfiguration.getLogin());
parser.setPassword(ExamplesConfiguration.getPassword());
parser.setPasswordEnable(ExamplesConfiguration.getPasswordEnable());

//ECP Operation Engine instantiation and configuration
cliCommands = parser.parseTemplate(ExamplesConfiguration.getTemplate());
//Equipment and Protocol Drivers
cliCommands.setProtocol(ExamplesConfiguration.getProtocol());
cliCommands.setConnectionResourceClassName(
    ExamplesConfiguration.getConnectionResourceClassName());

//Equipment Driver additional initialization parameters
HashMap oAddParams= new HashMap();
oAddParams.put(
    ConnectionResource.DefaultParameterNames.specificParameters,
    ExamplesConfiguration.getAdditionalData());
cliCommands.setEquipmentDriverAdditionalParameters(oAddParams);

//Operation execution
oRet=cliCommands.execute();
//Other possible executions would have been:
//oRet=cliCommands.executeActivation();
//oRet=cliCommands.revert();
//oRet=cliCommands.inverseActivation();

//Execution Output
System.out.println("RESULT HASHMAP:");
System.out.println(oRet);
System.out.println("COMMANDS SENT:");
System.out.println(cliCommands.getCommandsSent());
System.out.println("STDOUT:");
System.out.println(cliCommands.getStdOut());
}
}
```

To execute the example, run the following command, where <classpath> should contain all the libraries contained in the directory \$ECP_LIB plus the path where these classes have been compiled plus the path where the Equipment Driver libraries are located. The java version must be 1.4.2.

```
java -classpath <classpath> -Dactivator.dir.config=$ECP ETC
com.hp.spain.connection.pool.examples.DirectConnExample
```

3.4. Monitoring ECP through JMS

ECP can be monitored through JMS. JMS is a specification which defines a messaging API. Two version of the specification have been produced so far: 1.1 and the now obsolete 1.0.2b.

Depending on your system, you might have to use JMS version 1.0.2b or 1.1. For example, JBoss-4.x supports the JMS1.1 version of the specification, while JBoss-3.2.x supports JMS1.0.2b. From 3.2.8, JBoss also supports JMS1.1. If your system does not impose a JMS version, version 1.1 is recommended. JMS 1.1 is backwards-compatible that is, a JMS 1.0.2b client will work with a JMS 1.1 provider and a JMS 1.1 provider will work as a JMS 1.0.2b provider.

3.4.1. Including Additional Data in Activation JMS Messages:

JMS Activation monitoring messages won't be sent unless the client issuing the activation establishes some data to be included in the messages. When receiving the JMS messages through a JMS client, the data established by the ECP client will be received. This provides a way for the ECP client to communicate with

the JMS Client. The JMS Client will typically use this information to filter the messages it will receive (see JMS Documentation for additional information on this issue).

The following example shows how to establish the data to be sent in the messages.

```
package com.hp.spain.connection.pool.examples;

import java.util.HashMap;
import java.util.Map;

import com.hp.spain.connection.CLICommands;
import com.hp.spain.connection.CLIExecutionException;
import com.hp.spain.connection.TemplateParser;
import com.hp.spain.connection.TemplateParserException;
import com.hp.spain.connection.configuration.ECPSendingMessageConfiguration;

public class JMSMessagesActivationExample {

    public static void main (String[] args) throws CLIExecutionException, TemplateParserException
    {
        HashMap oRet=null; //the Operation execution result
        TemplateParser parser; //the ECP Template Parser instance
        CLICommands cliCommands; //the ECP Operation Engine instance

        //ECP Template Parser instantiation and configuration
        parser=new TemplateParser();

        //ECP Operation Engine instantiation and configuration
        cliCommands = parser.parseTemplate(ExamplesConfiguration.getTemplate());

        //ECP instance
        cliCommands.setRMISHostName(ExamplesConfiguration.getECPHost());
        cliCommands.setRMISPort(ExamplesConfiguration.getECPPort());

        //Set the content to include in the JMS Monitoring messages
        Map messagesConfiguration=new HashMap(); //The messages configuration
        //establish the messages configuration
        cliCommands.setMsgsSpecifier(messagesConfiguration);
        Map messagesAdditionalContents=new HashMap(); //The messages additional data
        //Include the additional message contents in the messages configuration.
        messagesConfiguration.put(
            ECPSendingMessageConfiguration.MSGSPEC_PROPID_JMSPROPERTIES,
            messagesAdditionalContents);

        //Add the messages additional data
        messagesAdditionalContents.put("par1", new Integer(1));
        messagesAdditionalContents.put("par2", "val2");

        //Operation execution
        oRet=cliCommands.execute(
            ExamplesConfiguration.getPoolName(),
            ExamplesConfiguration.getQueueID());

        //Execution Output
        System.out.println("RESULT HASHMAP:");
        System.out.println(oRet);
        System.out.println("COMMANDS SENT:");
        System.out.println(cliCommands.getCommandsSent());
        System.out.println("STDOUT:");
        System.out.println(cliCommands.getStdOut());
    }
}
```

3.4.2. JMS Client Dependencies

3.4.2.1. Integrating with another JMS provider

If your system does impose a JMS version (usually because it provides a JMS implementation), you will have to include the following library:

```
activemq-core-4.1.1.jar
```

The previous jar has a runtime dependency with the following jars. You may also have to include them if your system does not.

```
backport-util-concurrent-2.1.jar  
commons-logging-1.1.jar  
geronimo-j2ee-management_1.0_spec-1.0.jar
```

Those libraries provide the Active MQ 4.1.1 implementation of JMS, but do not include a JMS definition. As the JMS API version is imposed by your system, you should include one of your system libraries to provide the API definition. Check your system documentation to know which library to include. JBoss provides the following jars:

```
jboss-j2ee.jar  
jbossall-client.jar
```

Both jars include the JMS API definition. Use whichever you find more convenient, but not both.

3.4.2.2. No other JMS provider

If your system does not impose a JMS version (it does not include at least a runtime JMS API definition), you may use the JMS API version provided by Active MQ 4.1.1. You will have to include the following library

```
apache-activemq-4.1.1.jar
```

3.4.3. JMS Client Examples

What follows is a series of examples of JMS clients which work as ECP Monitors. In those examples, the following class, simulating a configuration, is used. You will probably have some other particular way of obtaining the configuration.

```
package com.hp.spain.connection.pool.examples;  
  
import java.util.Hashtable;  
  
import javax.jms.Session;  
import javax.naming.Context;  
  
public class JMSClientConfiguration {  
  
    private static final Hashtable contextEnvironment;  
  
    private static final boolean administeredConnectionFactory;  
    private static final String connectionFactoryJNDIName;  
    private static final String connectionFactoryURL;  
  
    private static final boolean transactedSession;  
    private static final int acknowledgeMode;  
  
    private static final String destinationJNDIName;  
  
    private static final String consumerMessagesFilter;  
    private static final boolean receiveLocalMessages;  
  
    private static final long messageReceptionTimeout;  
  
    static {  
        contextEnvironment = new Hashtable();  
        contextEnvironment.put(  
            Context.INITIAL_CONTEXT_FACTORY,  
            "org.apache.activemq.jndi.ActiveMQInitialContextFactory");  
        contextEnvironment.put(Context.PROVIDER_URL, "tcp://pallanthas.des.hp.es:4001");  
    }  
}
```

```

        administeredConnectionFactory = true;
        connectionFactoryJNDIName = "TopicConnectionFactory";
        connectionFactoryURL = "tcp://pallanthas.des.hp.es:4001";

        transactedSession=false;
        acknowledgeMode = Session.AUTO_ACKNOWLEDGE;

        destinationJNDIName = "dynamicTopics/ECP.MainTopic";

        consumerMessagesFilter = null;
        receiveLocalMessages = true;

        messageReceptionTimeOut = 10000;
    }

    public static String getConsumerMessagesFilter() {
        return consumerMessagesFilter;
    }

    public static boolean isReceiveLocalMessages() {
        return receiveLocalMessages;
    }

    public static boolean isAdministeredConnectionFactory() {
        return administeredConnectionFactory;
    }

    public static String getConnectionFactoryJNDIName() {
        return connectionFactoryJNDIName;
    }

    public static String getConnectionFactoryURL() {
        return connectionFactoryURL;
    }

    public static Hashtable getContextEnvironment() {
        return contextEnvironment;
    }

    public static int getAcknowledgeMode() {
        return acknowledgeMode;
    }

    public static boolean isTransactedSession() {
        return transactedSession;
    }

    public static String getDestinationJNDIName() {
        return destinationJNDIName;
    }

    public static long getMessageReceptionTimeOut() {
        return messageReceptionTimeOut;
    }
}
};

```

3.4.3.1. JMS 1.0.2b Client Example

```

package com.hp.spain.connection.pool.examples;

import javax.jms.MapMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.jms.TopicSubscriber;
import javax.naming.InitialContext;

public class JMS102bClient {
    public static void main(String[] args) throws Exception{
        InitialContext context;
        TopicConnectionFactory connectionFactory;
    }
}

```



```

TopicConnection connection;
TopicSession session;
Topic destination;
TopicSubscriber messageConsumer;
MapMessage message;

//naming context for administered objects
context = new InitialContext(JMSClientConfiguration.getContextEnvironment());

//the connection factory is obtained
connectionFactory = (TopicConnectionFactory) context.lookup(
    JMSClientConfiguration.getConnectionFactoryJNDIName());

//the connection is created
connection = connectionFactory.createTopicConnection();
//the session is created
session = connection.createTopicSession(
    JMSClientConfiguration.isTransactedSession(),
    JMSClientConfiguration.getAcknowledgeMode());

//the destination is obtained
destination = (Topic) context.lookup(JMSClientConfiguration.getDestinationJNDIName());

//the message receiver is created
messageConsumer = session.createSubscriber(
    destination,
    JMSClientConfiguration.getConsumerMessagesFilter(),
    JMSClientConfiguration.isReceiveLocalMessages());

//start to receive messages
connection.start();

//wait for a message
System.out.println("Waiting for message.");
Message message = null;
message = (MapMessage)messageConsumer.receive(
    JMSClientConfiguration.getMessageReceptionTimeout());
if (message!=null){
    //Process the message
    System.out.println("Received message: " + message.toString());

    //Acknowledge the message.
    //Acknowledging a consumed message acknowledges all messages that the session has
    //consumed.
    //This call can be omitted for both transacted sessions and sessions specified to use
    //implicit acknowledgement modes. However, extra care must be taken when omitting message
    //acknowledgement as messages that have been received but not acknowledged may be
    //redelivered.
    //Additionally, when client acknowledgment mode is used, a client may build up a large
    //number of unacknowledged messages while attempting to process them.
    //This call can be made before processing the message, if message losses are tolerated.
    message.acknowledge();
}
else {
    System.out.println("No message was received.");
}

//clean up
messageConsumer.close();
connection.stop();
session.close();
connection.close();

System.out.println("FINISHED.");
}
}

```

3.4.3.2. JMS 1.1 Client Example

```
package com.hp.spain.connection.pool.examples;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.MapMessage;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.naming.InitialContext;

public class JMS11Client {
    public static void main(String[] args) throws Exception{
        InitialContext context;
        ConnectionFactory connectionFactory;
        Connection connection;
        Session session;
        Destination destination;
        MessageConsumer messageConsumer;
        MapMessage message;

        //naming context for administered objects
        context = new InitialContext(JMSClientConfiguration.getContextEnvironment());

        //the connection factory is obtained
        connectionFactory = (ConnectionFactory) context.lookup(
            JMSClientConfiguration.getConnectionFactoryJNDIName());

        //the connection is created
        connection = connectionFactory.createConnection();
        //the session is created
        session = connection.createSession(
            JMSClientConfiguration.isTransactedSession(),
            JMSClientConfiguration.getAcknowledgeMode());

        //the destination is obtained
        destination = (Destination) context.lookup(
            JMSClientConfiguration.getDestinationJNDIName());

        //the message receiver is created
        messageConsumer = session.createConsumer(
            destination,
            JMSClientConfiguration.getConsumerMessagesFilter(),
            JMSClientConfiguration.isReceiveLocalMessages());

        //start to receive messages
        connection.start();

        //wait for a message
        System.out.println("Waiting for message.");
        message= null;
        message= (MapMessage)messageConsumer.receive(
            JMSClientConfiguration.getMessageReceptionTimeOut());
        if (message!=null){
            //process the message
            System.out.println("Received message: " + message.toString());

            //Acknowledge the message.
            //Acknowledging a consumed message acknowledges all messages that the session
            //has consumed.
            //This call can be omitted for both transacted sessions and sessions specified
            //to use implicit acknowledgement modes. However, extra care must be taken when
            //omitting message acknowledgement as messages that have been received but not
            //acknowledged may be redelivered.
            //Additionally, when client acknowledgment mode is used, a client may build up a
            //large number of unacknowledged messages while attempting to process them.
            //This call can be made before processing the message, if message losses are
            //tolerated.
            message.acknowledge();
        }
        else {
            System.out.println("No message was received.");
        }
    }
}
```

```

    }

    //clean up
    messageConsumer.close();
    connection.stop();
    session.close();
    connection.close();

    System.out.println("FINISHED.");
}
}

```

3.4.3.3. Processing Additional Data Included In Activation JMS Messages

When processing a JMS message, the additional data included by the client who issued the activation can be extracted by the JMS Client. The additional data is contained as named values inside the `MapMessage`. Extracting the data is a simple process:

```
message.getString("par");
```

3.4.4. ECP Messages Types

To ease client implementation, JMS provides the means to filter the messages that a `MessageConsumer/TopicSubscriber` will receive. See the JMS documentation for further details.

ECP Messages will always be instances of `MapMessage`.

All messages will contain a Header Property, with the name

```
com.hp.spain.connection.monitor.messages.ECPMessage.EventIDField.Name
```

and whose value will identify the type of message. The information available in a message will vary, depending on the type of message.

3.4.4.1. DataSent Message

If the message header property of name

```
com.hp.spain.connection.monitor.messages.ECPMessage.EventIDField.Name
```

has the value

```
com.hp.spain.connection.monitor.messages.DataSentMessage.EventIDField.Values.DataSent
```

the message is a `DataSent Message`. That type of message will be sent every time the protocol driver is instructed to send data to the equipment.

The message header property of name

```
com.hp.spain.connection.monitor.messages.DataSentMessage.EventDataField.Name
```

will contain the data sent.

3.4.4.2. DataReceived Message

If the message header property of name

```
com.hp.spain.connection.monitor.messages.ECPMessage.EventIDField.Name
```

has the value

```
com.hp.spain.connection.monitor.messages.DataReceivedMessage.EventID.Values.DataReceived
```

the message is a DataSent Message. Than type of message will be sent every time the protocol driver is instructed to receive data from the equipment.

The message header property of name

```
com.hp.spain.connection.monitor.messages.DataReceivedMessage.EventDataField.Name
```

will contain the data received.

3.5. Retrieving connection information through an external data getter

Since EP version 5.1-10, new parameters `ExternalDataGetterClassName` and `ElementName` have been added to subpools to allow defining an external data getter that will be used to retrieve connection information such as the equipment ip, port, protocol, etc. The value for the `ExternalDataGetterClassName` parameter should be the fully qualified name of a class implementing the `com.hp.spain.connection.ExternalDataGetter` interface:

```
public interface ExternalDataGetter
{
    public void setDBManager(DBManager dbManager);
    public ExternalData getExternalData(String elementName);
}
```

The `getExternalData` method is invoked by the subpool when creating a new connection resource and should return a new `com.hp.spain.connection.ExternalData` instance with the connection information for the given network element name. The implementer class can use the provided `DBManager` (`setDBManager` will be invoked before `getExternalData` with a proper one) to retrieve this connection information from the database, or otherwise guess it from anywhere else.

`ExternalData` contains the following fields:

```
public class ExternalData
{
    private String host;
    private String password;
    private String passwordEnable;
    private Integer port;
    private String protocol;
    private String user;
    private String userEnable;

    [...]
}
```

4. Configuration

Some of the configuration parameters will affect multiple ECP entities. As a consequence, it is recommended to check the indicated cross references to avoid collateral effects when modifying a parameter.

4.1. Common Configuration Sources

4.1.1. ProtocolDrivers.lst File

This file will configure the ProtocolDrivers to register and use (see Protocol Drivers Manager Configuration). The `ProtocolDrivers.lst` file should be located in the path `<ecp_home>\conf` by default, being `<ecp_home>` the ECP installation directory (see ECP RMI Service Command Line Parameters). The default location may be overwritten through the system property `activator.dir.config` (see ECP RMI Service Command Line Parameters). The file specifies the protocol driver classes, containing a single string with the following syntax:

```
<protocol_driver_list>:=<protocol_driver>{<sep><protocol_driver>}
<sep>:=,|:|;
```

where `<protocol_driver>` is the fully qualified name of the protocol driver class. It must implement `com.hp.spain.connection.ProtocolDriver`.

4.1.2. Protocol Drivers Configuration

Specific protocol driver configuration files can be found in the `$ECP_HOME/properties` directory. For example, the `TcpDriver.properties` file contains properties related to the operation of the TCP equipment driver.

4.1.2.1. TcpDriver Configuration

The following properties may be defined in `TcpDriver.properties` to customize `TcpDriver` behavior:

`bind_target0..N`: A destination specification. Connections to matching IPs or hostnames will be bound to the address specified by the corresponding `bind_address` parameter. The following target definition formats are supported:

- An IP address. Example: `1.2.3.4`
- A hostname. Example: `hp.com`
- CIDR notation. Example: `198.51.100.0/24`
- Dot-decimal notation. Example: `198.51.100.0/255.255.255.0`

`bind_address0..N`: The local IP address outgoing sockets should be bound to when connecting to destinations matching the corresponding bind target.

Example usage of the `bind_address` and `bind_target` properties:

```
bind_target0=198.51.100.0/24
bind_address0=198.51.100.2
```

```
bind target1=hp.com
bind address1=10.0.0.2

bind target2=10.0.1.100
bind address2=10.0.0.3
```

4.1.2.2. TelnetDriver Configuration

The following properties may be defined in `TelnetDriver.properties` to customize `TelnetDriver` behavior:

`bind_target0..N`: A destination specification. Connections to matching IPs or hostnames will be bound to the address specified by the corresponding `bind_address` parameter. See 4.1.2.2 for additional information.

`bind_address0..N`: The local IP address outgoing sockets should be bound to when connecting to destinations matching the corresponding bind target. See 4.1.2.2 for additional information.

4.1.3. HPSA_ECPMESSAGESPATTERNS

`IDMESSAGE`: Message Identifier. Mandatory. The sequence `HPSA_ECPMESSAGESPATTERNS_SEQ` should be used to establish the values of this field.

`CONNECTIONRESOURCECLASSNAME`: Canonical name of the equipment driver class to which the pattern applies. `null` if the pattern should be applied to all the drivers (and the protocol indicated by `PROTOCOL`).

`PROTOCOL`: Identifier of the protocol to which the pattern applies. `null` if the pattern should be applied to all the protocols (and the driver indicated by `CONNECTIONRESOURCECLASSNAME`).

`TYPE`: Reserved. Always `null`. In a future this field might be use to further restrict the scope of the pattern, i.e.: failures, errors...

`RESPONSEPATTERN`: Regular expression to be used to identify the message to return and to generate that message, as defined in Java 1.4 `java.util.regex.Pattern`. If the command response matches the pattern, the message generated will contain the command response with all the matches replaced with the replacement established in `responseReplacement`. Mandatory.

`RESPONSEREPLACEMENT`: Replacement value as defined in Java 1.4 `java.util.regex.Matcher#appendReplacement(StringBuffer, String)` which will be used to replace all the matches of `responsePattern` (if any) in the generated message. Mandatory.

4.1.4. HPSA_ECPCOMMANDSPATTERNS

`IDCOMMAND`: Command Identifier. Mandatory. The sequence `HPSA_ECPCOMMANDSPATTERNS_SEQ` should be used to stablish the values of this field.

`TYPE`: Reserverd. Always `null`. In a future this field might be use to further restrict the scope of the pattern, i.e.: failures, errors...

`COMMANDPATTERN`: Regular expression to be used to identify the message to return and to generate that message, as defined in Java 1.4 `java.util.regex.Pattern`. If the associated command response is matched, the command will be matched with this pattern and all the matches (if any) will be replaced by the replacement established in `commandReplacement`. Mandatory.

COMMANDREPLACEMENT: Replacement value as defined in Java 1.4

java.util.regex.Matcher#appendReplacement(StringBuffer, String) which will be used to replace all the matches of commandPattern (if any) in the generated message. Mandatory.

4.1.5. HPSA_ECPMESSAGESCOMMANDS

IDMESSAGE: Message Identifier. Mandatory.

IDCOMMAND: Command Identifier. Mandatory.

4.2. ECP Lib Configuration Sources

4.2.1. ECP Lib Command Line Parameters

ECP Lib uses the following JVM command line parameters:

```
-Dactivator.dir.config=<ecp_prot_drivers_dir>
```

<ecp_prot_drivers_dir>: Directory where the ProtocolDrivers.lst file can be found. This parameter is mandatory only if direct connections are used. In other case, it is not used.

4.3. ECP RMI Service Configuration Sources

4.3.1. ECP RMI Service Command Line Parameters

The command line of the ECP RMI Server JVM has the following syntax:

```
<java_exe> -server -Djava.rmi.server.codebase=file:<ecp_home>\rmi_pub  
-Djava.rmi.server.logCalls=false -Djava.rmi.server.hostname=<ecp_rmi_server_ip>  
-Djava.security.policy=<ecp_home>\conf\RmiEcpService.policy -  
Dactivator.dir.config=<ecp_prot_drivers_dir> -classpath <ecp_libs>  
com.hp.spain.connection.pool.server.RmiEcpService <ecp_rmi_registry_server_host>  
<ecp_rmi_registry_server_port> <ecp_home>
```

<java_exe>: path to the JVM executable file. Of course, it is mandatory.

<ecp_home>: ECP installation directory. This parameter is mandatory. It will be used to establish the ecp.properties and RmiEcpService.policy files location and the ProtocolDrivers.lst file default location. See ecp.properties File and ProtocolDrivers.lst File.

<ecp_rmi_server_ip>: IP of the localhost, used by the locally created stubs to access the RMI server. Used by the JVM. This parameter is mandatory.

<ecp_rmi_registry_server_host>: Host name of the host where the RMI registry is located and where the ECP RMI service object should be bound. Normally it should refer to the localhost. This parameter is mandatory.

<ecp_rmi_registry_server_port>: Port number where the RMI registry accepts calls and where the ECP RMI service object should be bound. This parameter is mandatory.

<ecp_libs>: all the .jar and .zip files in the directory <ecp_home>\lib. This parameter is mandatory.

<ecp_prot_drivers_dir>: Directory where the ProtocolDrivers.lst file can be found. This parameter is optional (see ProtocolDrivers.lst File).

Equipment Connection Pool

Note: when operating in cluster mode, parameters `ecp_rmi_server_ip` and `ecp_rmi_registry_server_host` should be configured so the ECP instance is reachable from other cluster members, so using 127.0.0.1 or localhost is not an option.`ecp.properties` File

The `ecp.properties` file should be located in the path `<ecp_home>\conf`, being `<ecp_home>` the ECP installation directory (See ECP RMI Service Command Line Parameters). The `ecp.properties` files may contain the following properties.

LOG_DIR: Logs directory. Most of the log data will be stored there. Its default value is `"C:\hp\OpenView\ServiceActivator\var\log"` in windows and `"/var/opt/OV/ServiceActivator/log/"` in HP-UX. It must end with the path separator character. This directory should exist and the user which executes the ECP RMI Service JVM must have writing permission over it. It will establish the Pool `LogFilePath` (see Pool Instance Specific Logging Parameters Configuration), `ProtocolDriver SpyFile` (see `ProtocolDriver` Configuration) and `Configurator Appender` (see `Configurator` Configuration).

LOG_MAX_FILE_SIZE: Will configure the `RollingFileAppenders` (when used) maximum file size (in bytes) before being rolled over to backup files. Its default value is 5242880 bytes (5MB). See Pool Logging Common Parameters Configuration and `Configurator` Configuration.

LOG_MAX_NUM_FILES: Will configure the `RollingFileAppenders` (when used) maximum backup index (how many backup files are kept). Its default value is 10. See Pool Logging Common Parameters Configuration and `Configurator` Configuration.

LOG_DATE_PATTERN: Will establish the type of `Appenders` used by the pools and configure the pools `DailyRollingFileAppenders` (when used) rolling date pattern. Its default value is `null`. It must be `null` or a valid `SimpleDateFormat` pattern (see <http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html>). See Pool Logging Common Parameters Configuration.

LOG_PATTERN: Will configure the messages pattern for the pools and the `Configurator`. Its default value is `null`. See Pool Logging Common Parameters Configuration and `Configurator` Configuration.

RELOAD_MAX_TIME: Will configure the subpools expiration timeout. Its default value is 0. (see `PoolManager` Configuration and ECP RMI Service)

MAX_REQUESTS_TO_GET_VERIFIED_RESOURCE: Will configure the maximum number of request to obtain a positively verified resource. Its default value is 1. Must be ≥ 1 . See Pool Common Parameters Configuration.

DEFAULT_QUEUE_ID: Will configure the default queue to add the resource requests to, if it is not specified or the specified queue is not found. Its default value is 1. See Pool Common Parameters Configuration.

MAX_POOLS: Will configure the maximum number of pools that can coexist simultaneously. Its default value is 0. See `PoolManager` Configuration.

DISPATCHER_MAX_RATE: Will configure the maximum number of connections assigned to the whole set of clients by second. Its default value is 10. See Pool Common Parameters Configuration.

RES_MGR_MAX_RATE: Will configure the maximum number of times per second that the expired resources will be finalized, the expired temporary resources deleted and the inactive resources reinitialized. Its default value is 1. See Pool Common Parameters Configuration.

REQ_MGR_MAX_RATE: Will configure the maximum number of times per second that the process of elimination and cancellation of expired resources requests will be executed. Its default value is 0.1. See Pool Common Parameters Configuration.

POOL_MGR_MAX_RATE: Will configure the number of times per second that the process of unloading dynamic expired pools will be executed. Its default value is 0.1. See PoolManager Configuration.

DYNAMIC_POOL_NOT_USED_MAX_TIME_LIFE: Will configure the default dynamic pools NotUsedMaxTimeLife. Its default value is 0. See Pool Instance Specific Parameters Configuration.

DYNAMIC_POOL_REQUEST_TIME_OUT: Will configure the dynamic pools RequestTimeout. Its default value is 0. See Pool Instance Specific Parameters Configuration.

DYNAMIC_POOL_NUM_QUEUES: Will configure the dynamic pools NumQueues. Its default value is 0. See Pool Instance Specific Parameters Configuration.

DYNAMIC_POOL_WEIGHT_QUEUES: Will configure the dynamic pools WeightQueues. Its default value is null. See Pool Instance Specific Parameters Configuration.

DYNAMIC_POOL_LOG_LEVEL: Will configure the dynamic pools LogLevel. Its default value is 0. See Pool Instance Specific Logging Parameters Configuration.

DYNAMIC_POOL_INIT_SESSIONS: Will configure the dynamic SubPools default InitSessions. Must be an integer value. If it equals 0, then, false. In other case, true. Its default value is 0. See SubPool Configuration

SubPool Instance Specific Parameters Configuration.

DYNAMIC_POOL_MAX_SESSIONS: Will configure the dynamic SubPools default MaxSessions. Its default value is 0. See SubPool Configuration

SubPool Instance Specific Parameters Configuration.

DYNAMIC_POOL_MIN_SESSIONS: Will configure the dynamic SubPools default MinSessions. Its default value is 0. See SubPool Configuration

SubPool Instance Specific Parameters Configuration.

DYNAMIC_POOL_RESOURCE_TIME_OUT: Will configure the dynamic SubPools default ResourceTimeout. Its default value is 0. See SubPool Configuration

SubPool Instance Specific Parameters Configuration.

DYNAMIC_POOL_TEMPORARY_RESOURCES_TIME_OUT: Will configure the dynamic SubPools default TemporaryResourcesTimeout. Its default value is 0. See SubPool Configuration

SubPool Instance Specific Parameters Configuration.

DB_DRIVER: Fully qualified class name of a `java.sql.Driver` to load and register in the JDBC DriverManager. Its default value is null. See DBManager Configuration

DB_USER: The DataBase user on whose behalf the connection is being made. Its default value is null. See DBManager Configuration

DB_PASSWORD: The DataBase user password. Its default value is null. See DBManager Configuration

DB_URL: A JDBC DataBase URL with the form: `jdbc:<subprotocol>:<subname>`. Its default value is null. See DBManager Configuration

CLUSTERED: Whether ECP operates in cluster mode (1) or not (0).

INSTANCE_NAME: The name of this particular instance in the cluster. If CLUSTERED is not defined or its value is 0, this property is ignored. Only alphanumeric characters, `_` and `-` are allowed in instance names.

`ECP.Msgs.Enable`: Whether the ECP should perform JMS monitoring or not. If this option is disabled, no JMS monitoring messages will be sent, and the JMS configuration parameters will be ignored. Its default value is "false".

`JMSBrokerReference.broker.uri`: URI of the JMS service where ECP JMS monitoring messages will be sent. Ignored if "`ECP.Msgs.Enable=false`". By default it will start an embedded JMS broker. Its default value is

`"vm\:(broker\:(tcp\://localhost\:4001)?brokerName\=EmbeddedBroker&useJmx\=true&persistent\=false&populateJMSXUserID\=false&useShutdownHook\=false&deleteAllMessagesOnStartup\=false&enableStatistics\=false)?marshal\=false"`.

`java.naming.factory.initial`: The Initial context factory for JMS Administered objects. Ignored if "`ECP.Msgs.Enable=false`". Its default value is

`"org.apache.activemq.jndi.ActiveMQInitialContextFactory"`.

`JMSMessageBroker.dest.type`: The type of the JMS destination where the ECP JMS Monitoring messages will be sent. Use "temp" to indicate a temporary Destination and "administered" to indicate an administered one. Ignored if "`ECP.Msgs.Enable=false`". Its default value is "administered".

`JMSMessageBroker.dest.name`: The JMS destination where the ECP JMS Monitoring messages will be sent. If the destination type in "`JMSMessageBroker.dest.type`" is temporary, any value will suffice; if the destination type in "`JMSMessageBroker.dest.type`" is administered, this property must contain the name under which the Destination is registered. Ignored if "`ECP.Msgs.Enable=false`". Its default value is "ECP.MainTopic".

4.3.2. ecp-cluster.properties

A separate `ecp-cluster.properties` file may be placed in `$ECP_HOME/conf` to fine-tune some aspects of cluster operation (when `CLUSTERED=1` in `ecp.properties`).

The following properties are supported:

`heartbeat.interval`: Time in ms between heartbeat updates. Default is 30000.

`heartbeat.threshold`: Time in ms after which an ECP instance that has not updated its heartbeat record must be considered to be dead. Default is 50000.

`heartbeat.max.errors`: Number of consecutive failures to update heartbeat after which the ECP instance shuts itself down. Default is 2.

`monitor.interval`: Time in ms between cluster monitor iterations to check for new/deleted pools, new/dead ECP instances, configuration changes, etc. Default is 30000.

`monitor.grace.period`: Amount of time in ms before an ECP instance begins taking over pools from dead cluster members that have preference over it for those pools. Default is 120000.

`loader.consumers`: Number of parallel pool loader threads. Default is 5.

`loader.lock.max.time`: Time in ms an ECP instance has to unload a pool that is being taken back before force-finishing active connections. Default is 600000.

4.3.3. HPSA_EQUIPMENTCONNPOOL DB Table

`NAME`: Will configure the static pool Name. See Pool Instance Specific Parameters Configuration.

`NOTUSEDMAXTIMELIFE`: Will configure the static pool `NotUsedMaxTimeLife`. See Pool Instance Specific Parameters Configuration.

REQUESTTIMEOUT: Will configure the static pool `RequestTimeout`. See [Pool Instance Specific Parameters Configuration](#).

NUMQUEUES: Will configure the static pool `NumQueues`. See [Pool Instance Specific Parameters Configuration](#).

WEIGHTQUEUES: Will configure the static pool `WeightQueues`. See [Pool Instance Specific Parameters Configuration](#).

CLUSTERED: Will configure the static pool `Clustered`. See [Pool Instance Specific Parameters Configuration](#).

INSTANCE_NAMES: Will configure the static pool `InstanceNames`. See [Pool Instance Specific Parameters Configuration](#).

NUM_INSTANCES: Will configure the static pool `NumInstances`. See [Pool Instance Specific Parameters Configuration](#).

LOGFILE: Will configure the static pool `LogFilePath`. See [Pool Instance Specific Logging Parameters Configuration](#)

LOGLEVEL: Will configure the static pool `LogLevel`. See [Pool Instance Specific Logging Parameters Configuration](#).
HPSA_EQUIPMENTCONNSUBPOOL DB Table

INITSESSIONS: Will configure the static subpool `InitSessions`. See [SubPool Configuration SubPool Instance Specific Parameters Configuration](#).

MAXSESSIONS: Will configure the static subpool `MaxSessions`. See [SubPool Configuration SubPool Instance Specific Parameters Configuration](#).

MINSESSIONS: Will configure the static subpool `MinSessions`. See [SubPool Configuration SubPool Instance Specific Parameters Configuration](#).

RESOURCE_TIMEOUT: Will configure the static subpool `ResourceTimeout`. See [SubPool Configuration SubPool Instance Specific Parameters Configuration](#).

TEMPORARYRESOURCE_TIMEOUT: Will configure the static subpool `TemporaryResourcesTimeout`. See [SubPool Configuration SubPool Instance Specific Parameters Configuration](#).

NAMEPOOL: The register in the table `HPSA_EQUIPMENTCONNPOOL` associated with this one.

IDSUBPOOL: Will configure the static subpool `Id`. See [SubPool Configuration SubPool Instance Specific Parameters Configuration](#).

CONNECTIONRESOURCECLASSNAME: Will configure the static subpool `ConnectionResourceClassName`. See [SubPool Configuration SubPool Instance Specific Parameters Configuration](#).

IP: Will configure the static subpool `EquipmentDrivers` initialization parameter `IP`. See [EquipmentDriver Initialization Parameters Configuration](#).

PORT: Will configure the static subpool EquipmentDrivers initialization parameter `Port`. See EquipmentDriver Initialization Parameters Configuration.

PROTOCOL: Will configure the static subpool EquipmentDrivers initialization parameter `Protocol`. See EquipmentDriver Initialization Parameters Configuration.

USERNAME: Will configure the static subpool EquipmentDrivers initialization parameter `Username`. See EquipmentDriver Initialization Parameters Configuration.

PASSWORD: Will configure the static subpool EquipmentDrivers initialization parameter `Password`. See EquipmentDriver Initialization Parameters Configuration.

USERENABLE: Will configure the static subpool EquipmentDrivers initialization parameter `Userenable`. See EquipmentDriver Initialization Parameters Configuration.

PASSWORDENABLE: Will configure the static subpool EquipmentDrivers initialization parameter `Passwordenable`. See EquipmentDriver Initialization Parameters Configuration.

ELEMENTNAME: Will configure the static subpool element name, used by `ExternalDataGetter` implementations to retrieve the equipment connection information.

EXTERNALDATAGETTERCLASSNAME: Will configure the static subpool `ExternalDataGetterClassName`. See SubPool Configuration

4.3.4. SubPool Instance Specific Parameters Configuration.

4.3.4.1. DynamicECPPProperties Class

This class stores the configuration of a dynamic `Pool` and a `SubPool`. A dynamic `Pool` will always contain a single `SubPool`.

4.3.4.2. DynamicECPPProperties Properties

vi. DynamicECPPProperties Pool Properties

PoolName: Will configure the dynamic pool `Name` (see Pool Instance Specific Parameters Configuration) and `LogFilePath` (Pool Instance Specific Logging Parameters Configuration).

Clustered: Will configure the dynamic pool `Name` (see Pool Instance Specific Parameters Configuration).

InstanceNames: Will configure the dynamic pool `InstanceNames` (see Pool Instance Specific Parameters Configuration).

NumInstances: Will configure the dynamic pool `NumInstances` (see Pool Instance Specific Parameters Configuration).

vii. DynamicECPPProperties SubPool Properties

ConnectionResourceClassName: Will configure the dynamic subpool `ConnectionResourceClassName`. See SubPool Configuration

SubPool Instance Specific Parameters Configuration.

IP: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Ip`. See EquipmentDriver Initialization Parameters Configuration.

Port: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Port`. See EquipmentDriver Initialization Parameters Configuration.

Protocol: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Protocol`. See EquipmentDriver Initialization Parameters Configuration.

User: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Username`. See EquipmentDriver Initialization Parameters Configuration.

Password: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Password`. See EquipmentDriver Initialization Parameters Configuration.

UserEnable: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Userenable`. See EquipmentDriver Initialization Parameters Configuration.

PasswordEnable: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Passwordenable`. See EquipmentDriver Initialization Parameters Configuration.

ElementName: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Elementname`. See EquipmentDriver Initialization Parameters Configuration.

ExternalDataGetterClassName: Will configure the dynamic subpool EquipmentDrivers initialization parameter `Externaldatagetterclassname`. See EquipmentDriver Initialization Parameters Configuration.

4.3.4.3. DynamicECPPProperties Advanced Properties

viii. DynamicECPPProperties Advanced Pool Properties

NotUsedMaxTimeLife: Will configure the pool `NotUsedMaxTimeLife`. See Pool Instance Specific Parameters Configuration.

ix. DynamicECPPProperties Advanced SubPool Properties

InitSessions: Will configure the subpool `Initsessions`. See SubPool Configuration SubPool Instance Specific Parameters Configuration.

MaxSessions: Will configure the subpool `MaxSessions`. See SubPool Configuration SubPool Instance Specific Parameters Configuration.

MinSessions: Will configure the subpool `MinSessions`. See SubPool Configuration SubPool Instance Specific Parameters Configuration.

ResourceTimeout: Will configure the subpool `ResourceTimeout`. See SubPool Configuration SubPool Instance Specific Parameters Configuration.

TemporaryResourcesTimeout: Will configure the subpool `TemporaryResourcesTimeout`. See SubPool Configuration SubPool Instance Specific Parameters Configuration.

5. Commands Template Reference

5.1. Commands Template Commands

5.1.1. Block declaration Statements

Block Declaration Statements are those which indicate the beginning or ending of a Block. The following are Block Declaration Statements

```
[TEMPLATE:Config]
[TEMPLATE:Do]
[TEMPLATE:ErrorSection]
[TEMPLATE:Undo]
[TEMPLATE:Commit]
[TEMPLATE:Rollback]
[TEMPLATE:Section *]
```

Block Declaration Statements may include other Block Declaration Statements (depending on the containing statement) or Executable Statements

5.1.2. Executable Statements

Executable Statements are those which do not intervene in the Declaration of a Block but are included by them and define commands to be executed. They must always appear inside a Block.

5.1.2.1. If-Else Statement

Declares two different Executable blocks to execute depending on the value of a condition. The “Else” part is optional. It is defined like this:

```
[TEMPLATE:If "<condition>"]
  <Executable Statement Block>
[TEMPLATE:Else]
  <Executable Statement Block>
[TEMPLATE:EndIf]
```

Where <condition> is any valid ECP condition and <Executable Statement Block> is a set of Executable Statements. Nested If-Else or ForEach Statements are allowed. For example:

```
[TEMPLATE:If "failed=="true""]
  help
    [TEMPLATE:EndStrPattern "admin#"]
  exit
    [TEMPLATE:EndStrPattern "admin#"]
[TEMPLATE:EndIf]
```

Or with an else clause:

```
[TEMPLATE:If "failed=="true""]
  help
    [TEMPLATE:EndStrPattern "admin#"]
[TEMPLATE:Else]
  telnet 127.0.0.1 1234
    [TEMPLATE:EndStrPattern "admin#"]
[TEMPLATE:EndIf]
```

5.1.2.2. ForEach Statement

It declares an executable block to be executed once for every element of an Array Variable. It is defined like this:

```
[TEMPLATE:ForEach "<variableID>" In "<arrayVariableID>"]
  <Executable Statement Block>
```

```
[TEMPLATE:EndFor]
```

Where <variableID> is any valid ECP variable identifier, <arrayVariableID> is any valid ECP array variable identifier and <Executable Statement Block> is a set of Executable Statements.

On each loop, the variable <variableID> will contain a different value in <arrayVariableID> and its values will be in the same order as in <arrayVariableID>. Nested If-Else or ForEach Statements are allowed. For example:

```
[TEMPLATE:ForEach "var" In " destinationIPs"]
  ping %var% -n 1
  [TEMPLATE:EndStrPattern "admin#"]
[TEMPLATE:EndFor]
```

5.1.3. Command Statements

Command Statements are those which define how a command must be issued and its output processed. The following are Command Statements

```
[TEMPLATE:ErrorMessage "***"]
[TEMPLATE:NonError "***"]
[TEMPLATE:NonErrorPattern "***"]
[TEMPLATE:Error "***"]
[TEMPLATE:Failure "***"]
[TEMPLATE:EndStr "***"]
[TEMPLATE:EndStrPattern "***"]
[TEMPLATE:Secret]
[TEMPLATE:Echo]
[TEMPLATE:EndParamString "***"]
[TEMPLATE:EndCommandString "***"]
[TEMPLATE:Question "*** Response ***"]
[TEMPLATE:Filename "*** Options ***"]
[TEMPLATE:Filename "***"]
[TEMPLATE:Pattern "***"]
[TEMPLATE:Condition "***"]
[TEMPLATE:ExecuteUntil "***"]
[TEMPLATE:IgnoreDefaultEndStr]
[TEMPLATE:CommandDelay "***"]
[TEMPLATE:ReadAttempts "***"]
[TEMPLATE:Variable "***"]
[TEMPLATE:Array "***"]
[TEMPLATE:ExecuteUntilDelay "***"]
[TEMPLATE:ExecuteUntilAttempts "***"]
```

5.2. Commands Reference

5.2.1. Commands List

| Token | Explanation |
|--|--|
| !<comment> | A coment line. Lines starting with ! will be ignored. |
| [TEMPLATE:DefaultDelay "***"] | Wait that number of milliseconds before issuing every command by default |
| [TEMPLATE:DefaultReadAttempts "***"] | Fail if that number of socket read operations return nothing on every command by default |
| [TEMPLATE:DefaultEndParamString "***"] | String to append to the question responses on every command by default |
| [TEMPLATE:DefaultEndCommandString "***"] | String to append to every command by default |

Equipment Connection Pool

| | |
|---------------------------------------|---|
| [TEMPLATE:DefaultError "*"] | Pattern to interpret as the command returning an error on every command by default |
| [TEMPLATE:DefaultNonError "*"] | Pattern to ignore on error search on every command by default (not on failure search) |
| [TEMPLATE:DefaultNonErrorPattern "*"] | Regular Expression Pattern to ignore on error search on every command by default (not on failure search) |
| [TEMPLATE:DefaultEndStr "*"] | Pattern to interpret as the end of the command execution on every command by default |
| [TEMPLATE:DefaultEndStrPattern "*"] | Regular Expression Pattern to interpret as the end of the command execution on every command by default |
| | |
| [TEMPLATE:DefaultEcho] | Target System will echo, treat it as if it never arrived on every command by default |
| | |
| [TEMPLATE:Config] | Declares the beginning of a Config Block |
| [TEMPLATE:Do] | Declares the beginning of a Do Block |
| [TEMPLATE:ErrorSection] | Declares the beginning of an Error Block |
| [TEMPLATE:Undo] | Declares the beginning of an Undo Block |
| [TEMPLATE:Commit] | Declares the beginning of a Commit Block |
| [TEMPLATE:Rollback] | Declares the beginning of a Rollback Block |
| | |
| [TEMPLATE:Section *] | Declares the beginning of a Section Block |
| | |
| <comando> | |
| [TEMPLATE:AssignVariable "*"] | Assigns a constant value to a variable |
| [TEMPLATE:If "*"] | Declares an Executable block to execute if the condition is true |
| [TEMPLATE:Else] | Declares an Executable block to execute if the corresponding if condition is false |
| [TEMPLATE:EndIf] | Declares the end of an if or else executable block |
| [TEMPLATE:ForEach "*" In "*"] | Declares an Executable block to be executed once for every element of an Array Variable |
| [TEMPLATE:EndFor] | Declares the end of a forEach executable block |
| | |
| [TEMPLATE:ErrorMessage "*"] | An error message to be sent to the ECP client if an error is encountered |
| [TEMPLATE:NonError "*"] | Pattern to ignore on error search (not on failure search) |
| [TEMPLATE:NonErrorPattern "*"] | Regular Expression Pattern to ignore on error search (not on failure search) |
| [TEMPLATE:Error "*"] | Pattern to interpret as the command returning an error (and wait for command response end string) |
| [TEMPLATE:Failure "*"] | Pattern to interpret as the command returning a failure (and do not wait for command response end string) |

| | |
|--------------------------------------|---|
| [TEMPLATE:EndStr "*"] | Pattern to interpret as the end of the command execution |
| [TEMPLATE:EndStrPattern "*"] | Regular Expression Pattern to interpret as the end of the command execution |
| [TEMPLATE:Secret] | Do not trace the issued command (useful for passwords) |
| [TEMPLATE:Echo] | Target System will echo, treat it as if never arrived |
| [TEMPLATE:EndParamString "*"] | String to append to the question responses |
| [TEMPLATE:EndCommandString "*"] | String to append to the command |
| [TEMPLATE:Question "*" Response "*"] | The command is interactive. If the Pattern is found, send that response |
| [TEMPLATE:Pattern "*"] | Store every occurrence of the group in the regular expression in a position of the array variables which follow |
| [TEMPLATE:Condition "*"] | Only issue the command if the condition is satisfied |
| [TEMPLATE:ExecuteUntil "*"] | Execute the command until the condition is satisfied |
| [TEMPLATE:CommandDelay "*"] | Wait that number of milliseconds before issuing the command |
| [TEMPLATE:ReadAttempts "*"] | Fail if that number of socket read operations return nothing. |
| [TEMPLATE:ErrorPattern "*\\""] | Regular Expression Pattern to interpret as the command returning an error |
| [TEMPLATE:FailurePattern "*\\""] | Regular Expression Pattern to interpret as the command returning a failure |
| [TEMPLATE:Variable "*"] | Variable where to store the occurrence of the Pattern |
| [TEMPLATE:Array "*"] | Array variable where to store the occurrences |
| [TEMPLATE:ExecuteUntilDelay "*"] | Wait that number of milliseconds before issuing the command on every iteration |
| [TEMPLATE:ExecuteUntilAttempts "*"] | If that number of iterations is exceeded, ignore condition and exit loop |
| [TEMPLATE:Filename "*"] | Save the output to the file |
| [TEMPLATE:Filename "*" Option "*"] | Save the output to the file. Options can be bigoutput to true/false , tail and grep. The format to define options are bigoutput=false;tail=100;grep=100 |

5.2.2. Commands Syntax

| Token | Unique in Context | Precedes to Token (and) | Follows to Token (or) | Nested to Token | Mutually exclusive with Token |
|---|-------------------|-------------------------|-----------------------|-----------------|-------------------------------|
| <comment> | | | | | |
| [TEMPLATE:DefaultDelay "**"] | Yes | [TEMPLATE:Config] | | | |
| [TEMPLATE:DefaultReadAttempts "**"] | | [TEMPLATE:Config] | | | |
| [TEMPLATE:DefaultEndParamString "**"] | Yes | [TEMPLATE:Config] | | | |
| [TEMPLATE:DefaultEndCommandString "**"] | Yes | [TEMPLATE:Config] | | | |
| | | | | | |

Equipment Connection Pool

| | | | | | |
|--------------------------------------|-----|---|--|---|---------------------------------------|
| [TEMPLATE:DefaultError ""] | | [TEMPLATE:Config] | | | [TEMPLATE:DefaultErrorNull] |
| [TEMPLATE:DefaultNonError ""] | | [TEMPLATE:Config] | | | [TEMPLATE:DefaultNonErrorNull] |
| [TEMPLATE:DefaultNonErrorPattern ""] | | [TEMPLATE:Config] | | | [TEMPLATE:DefaultNonErrorPatternNull] |
| [TEMPLATE:DefaultEndStr ""] | | [TEMPLATE:Config] | | | [TEMPLATE:DefaultEndStrNull] |
| [TEMPLATE:DefaultEndStrPattern ""] | | [TEMPLATE:Config] | | | [TEMPLATE:DefaultEndStrPatternNull] |
| | | | | | |
| [TEMPLATE:DefaultEcho] | | [TEMPLATE:Config] | | | |
| | | | | | |
| [TEMPLATE:Config] | Yes | [TEMPLATE:Do] [TEMPLATE:ErrorSection] [TEMPLATE:ErrorSection FinalCommand] [TEMPLATE:Undo] [TEMPLATE:Commit] [TEMPLATE:Commit FinalCommit] [TEMPLATE:Rollback] [TEMPLATE:Rollback FinalRollback] [TEMPLATE:Exit] [TEMPLATE:Exit FinalExit] | | | [TEMPLATE:Config FinalConfig] |
| [TEMPLATE:Do] | Yes | | | | |
| [TEMPLATE:ErrorSection] | Yes | | [TEMPLATE:Do] | | [TEMPLATE:ErrorSection FinalCommand] |
| [TEMPLATE:Undo] | Yes | | [TEMPLATE:Do] [TEMPLATE:ErrorSection] | | |
| [TEMPLATE:Commit] | | | [TEMPLATE:Undo] | | |
| [TEMPLATE:Rollback] | | | [TEMPLATE:Undo] [TEMPLATE:Commit] [TEMPLATE:Commit FinalCommit] | | |
| | | | | | |
| [TEMPLATE:Section *] | | | | [TEMPLATE:Do] [TEMPLATE:Undo] | |
| | | | | | |
| <comando> | | | | [TEMPLATE:Config] [TEMPLATE:Config FinalConfig] [TEMPLATE:ErrorSection] [TEMPLATE:ErrorSection FinalCommand] [TEMPLATE:Commit] [TEMPLATE:Commit FinalCommit] [TEMPLATE:Rollback] [TEMPLATE:Rollback FinalRollback] [TEMPLATE:Exit] [TEMPLATE:Exit FinalExit] [TEMPLATE:Do]/[TEMPLATE:Section *] [TEMPLATE:Undo]/[TEMPLATE:Section *] | |
| [TEMPLATE:AssignVariable ""] | | | | idem <comando> | |
| [TEMPLATE:If ""] | | | | idem <comando> | |

| | | | | | |
|------------------------------------|--|--|--|----------------------------|--|
| [TEMPLATE:Else] | | | | idem <comando> | |
| [TEMPLATE:EndIf] | | | | idem <comando> | |
| [TEMPLATE:ForEach "" In ""] | | | | idem <comando> | |
| [TEMPLATE:EndFor] | | | | idem <comando> | |
| | | | | | |
| [TEMPLATE:ErrorMessage ""] | | | | <comando> | |
| [TEMPLATE:NonError ""] | | | | <comando> | |
| [TEMPLATE:NonErrorPattern ""] | | | | <comando> | |
| [TEMPLATE:Error ""] | | | | <comando> | |
| [TEMPLATE:Failure ""] | | | | <comando> | |
| [TEMPLATE:EndStr ""] | | | | <comando> | |
| [TEMPLATE:EndStrPattern ""] | | | | <comando> | |
| [TEMPLATE:Secret] | | | | <comando> | |
| [TEMPLATE:Echo] | | | | <comando> | |
| [TEMPLATE:EndParamString ""] | | | | <comando> | |
| [TEMPLATE:EndCommandString ""] | | | | <comando> | |
| [TEMPLATE:Question "" Response ""] | | | | <comando> | |
| [TEMPLATE:Pattern ""] | | | | <comando> | |
| [TEMPLATE:Condition ""] | | | | <comando> | |
| [TEMPLATE:ExecuteUntil ""] | | | | <comando> | |
| [TEMPLATE:CommandDelay ""] | | | | <comando> | |
| [TEMPLATE:ReadAttempts ""] | | | | <comando> | |
| [TEMPLATE:ErrorPattern \[""]\] | | | | <comando> | |
| [TEMPLATE:FailurePattern \[""]\] | | | | <comando> | |
| | | | | | |
| [TEMPLATE:Variable ""] | | | | [TEMPLATE:Pattern ""] | |
| [TEMPLATE:Array ""] | | | | [TEMPLATE:Pattern ""] | |
| | | | | | |
| [TEMPLATE:ExecuteUntilDelay ""] | | | | [TEMPLATE:ExecuteUntil ""] | |
| [TEMPLATE:ExecuteUntilAttempts ""] | | | | [TEMPLATE:ExecuteUntil ""] | |

6. Configuration Quick Reference

6.1. DBManager Configuration

Driver: A `java.sql.Driver` to load (through a `Class.forName()`) for it to be registered in the JDBC `DriverManager`. The class must exist and be in the classpath. Established from the `ecp.properties` `DB_DRIVER` property (see `ecp.properties` File).

User: The DataBase user on whose behalf the connection is being made. Established from the `ecp.properties` `DB_USER` property (see `ecp.properties` File).

Password: The DataBase user password. Established from the `ecp.properties` `DB_PASSWORD` property (see `ecp.properties` File).

URL: A JDBC URL String in the form `"jdbc:<subprotocol>:<subname>"`. Established from the `ecp.properties` `DB_URL` property (see `ecp.properties` File).

6.2. Configurator Configuration

The following parameters of the `Configurator` may be established:

Appender: The `Configurator` Appender properties (except for the file name which is fixed) may be configured from the `ecp.properties` `LOG_DIR`, `LOG_MAX_FILE_SIZE` and `LOG_MAX_NUM_FILES` properties (see `ecp.properties` File).

The `Configurator` will use a `RollingFileAppender` as Appender. Its maximum file size and maximum number of files will be the values specified by `LOG_MAX_FILE_SIZE` and `LOG_MAX_NUM_FILES` respectively. The `Configurator` log file will be located at `LOG_DIR + "Configurator.log"`. See `ecp.properties` File.

Pattern: Established from the `ecp.properties` `LOG_PATTERN` property (see `ecp.properties` File). It will configure the log messages format pattern of the `Configurator`. Its valid values may be:

```

null
ISO8601

```

A valid `PatternLayout`'s pattern.

If the pattern is `null` or `"ISO8601"` a `TTCCLayout` will be used as the Appender Layout. In other case, a `PatternLayout` with specified value will be used. See

<http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/TTCCLayout.html>

<http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html>

6.3. ECP RMI Service

reloadSubPool: On a `reloadSubPool` call, the timeout given to the whole set of the `BUSY` resources of the subpool before forcing their finalization can be configured through the `ecp.properties` file `RELOAD_MAX_TIME` property. If its value is `==0`, the process will wait until the resources are not `BUSY`. See `ecp.properties` File.

lockSubPool: On a `lockSubPool` call, the timeout given to the whole set of the `BUSY` resources of the subpool before forcing their finalization can be configured through the `ecp.properties` file `RELOAD_MAX_TIME` property. If its value is `<=0`, no timeout will be given. See `ecp.properties` File.

lockPool: On a `lockPool` call, the timeout given to the whole set of the `BUSY` resources of each subpool before forcing their finalization can be configured through the `ecp.properties` file `RELOAD_MAX_TIME` property. If its value is ≤ 0 , no timeout will be given. See `ecp.properties` File.

unloadPool: On a `unloadPool` call, the timeout given to the whole set of the `BUSY` resources of each subpool before forcing their finalization can be configured through the `ecp.properties` file `RELOAD_MAX_TIME` property. If its value is ≤ 0 , no timeout will be given. See `ecp.properties` File.

6.4. PoolManager Configuration

Pool Expiration: When a (not `BUSY`) pool expires and is unloaded, the timeout in milliseconds given to each subpool for its `BUSY` resources before forcing their finalization can be configured through the `ecp.properties` file `RELOAD_MAX_TIME` property. If its value is 0, no timeout will be given. See `ecp.properties` File.

MaxPools: The maximum number of pools that can coexist simultaneously can be configured through the `ecp.properties` file `MAX_POOLS` property. If its value is 0, no limit will be established. See `ecp.properties` File.

PoolCleanUp: The number of times per second that the process of unloading dynamic expired pools will be executed can be configured through the `ecp.properties` file `POOL_MGR_MAX_RATE` property. Must be $\neq 0$. See `ecp.properties` File.

6.5. Pool Configuration

6.5.1. Pool Common Parameters Configuration

The following parameters are shared by all the pools:

getResourceRetries: The number of request to obtain a positively verified resource can be configured through the `ecp.properties` file `MAX_REQUESTS_TO_GET_VERIFIED_RESOURCE` property. See `ecp.properties` File.

getResourceDefaultQueueId: The default queue to add the resource request to, if it is not specified or the specified queue is not found can be configured through the `ecp.properties` file `DEFAULT_QUEUE_ID` property. See `ecp.properties` File.

DispatcherMaxRate: The maximum number of connections assigned to the whole set of clients by second can be configured through the `ecp.properties` file `DISPATCHER_MAX_RATE` property. Must be $\neq 0$. See `ecp.properties` File.

ResourcesCleanUp: The maximum number of times per second that the expired resources will be finalized, the expired temporary resources deleted and the inactive resources reinitialized can be configured through the `ecp.properties` file `RES_MGR_MAX_RATE` property. Must be $\neq 0$. See `ecp.properties` File.

RequestsCleanUp: The maximum number of times per second that the process of elimination and cancellation of expired resources requests will be executed can be configured through the `ecp.properties` file `REQ_MGR_MAX_RATE` property. Must be $\neq 0$. See `ecp.properties` File.

6.5.1.1. Pool Logging Common Parameters Configuration

Appender: Each pool will have its own Appender, but the Appenders properties and types are common, except for the file path (LogFile) which is specific for each pool, see Pool Instance Specific Logging Parameters Configuration. The pools Appenders types and properties may be configured from the ecp.properties LOG_MAX_FILE_SIZE, LOG_MAX_NUM_FILES and LOG_DATE_PATTERN properties (see ecp.properties File).

If LOG_DATE_PATTERN is null, then a RollingFileAppender will be used. In other case, a DailyRollingFileAppender will be used. If a DailyRollingFileAppender is used, its rolling date pattern will be the value specified by LOG_DATE_PATTERN. If a RollingFileAppender is used, its maximum file size and maximum number of files will be the values specified by LOG_MAX_FILE_SIZE and LOG_MAX_NUM_FILES respectively. See ecp.properties File. Notice that each SubPool will use the logger of the pool it belongs to, to log its messages (see SubPool Instance Specific Logging Parameters Configuration)

Pattern: Established from the ecp.properties LOG_PATTERN property (see ecp.properties File). Will configure the pools log messages format pattern. Its valid values may be:

```

null
ISO8601

```

A valid PatternLayout's pattern.

If the pattern is null or "ISO8601" a TTCCLayout will be used as the Appender Layout. In other case, a PatternLayout with specified value will be used. See

<http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/TTCCLayout.html>

<http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html>

Notice that each SubPool will use the logger of the pool it belongs to, to log its messages (see SubPool Instance Specific Logging Parameters Configuration)

6.5.2. Pool Instance Specific Parameters Configuration

Name: Establishes the name that will identify the pool. If the pool is dynamic, and a name has been set (see DynamicECPPProperties Pool Properties) the value of this parameter will be

```

PoolName + "-" + user + "-" + ip + "-" + port

```

If a name has not been set, the value of this parameter will be

```

"DYNAMIC" + "-" + user + "-" + ip + "-" + port

```

If the pool is static, the value of this parameter will be specified by the HPSA_EQUIPMENTCONNPOOL NAME DB field (see HPSA_EQUIPMENTCONNPOOL DB Table).

NotUsedMaxTimeLife: Establishes the maximum time that a pool may remain unused. Once that time has expired, it will be removed. The timer is reset on each Operation (execute, executeActivation, inverseActivation, revert). If the pool is dynamic, and the advanced dynamic properties are set (see DynamicECPPProperties Advanced Properties) the value of this parameter will be specified by the DynamicECPPProperties NotUsedMaxTimeLife attribute (see DynamicECPPProperties Advanced Pool Properties). If the advanced dynamic properties are not set, the value of this parameter will be specified by the ecp.properties DYNAMIC_POOL_NOT_USED_MAX_TIME_LIFE property (see ecp.properties File). If the pool is static, the value of this parameter will be specified by the HPSA_EQUIPMENTCONNPOOL NOTUSEDMAXTIMELIFE DB field (see HPSA_EQUIPMENTCONNPOOL DB Table).

RequestTimeout: Maximum time to wait when obtaining a connection on a client request.

If the pool is dynamic, the value of this parameter will be specified by the `ecp.properties` `DYNAMIC_POOL_REQUEST_TIME_OUT` property (see `ecp.properties` File). If the pool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNPOOL REQUESTTIMEOUT` DB field (see `HPSA_EQUIPMENTCONNPOOL` DB Table).

NumQueues: Number of request queues to add to the pool. Its value must be coherent with the value specified in **WeightQueues**.

If the pool is dynamic, the value of this parameter will be specified by the `ecp.properties` `DYNAMIC_POOL_NUM_QUEUES` property (see `ecp.properties` File). If the pool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNPOOL NUMQUEUES` DB field (see `HPSA_EQUIPMENTCONNPOOL` DB Table).

WeightQueues: Priority of each request queue. The number of request queues specified in **NumQueues** will be created with the specified corresponding weights and order, and with the ids from 1 to **NumQueues**. It must not be null, and must have the format:

```
<weight queues>:=<queue weight>{<sep><queue weight>}
<sep>:=,
```

Where `<queue_weight>` is a number specifying the weight of the queue. The higher the weight, the higher the priority of the queue.

If the pool is dynamic, the value of this parameter will be specified by the `ecp.properties` `DYNAMIC_POOL_WEIGHT_QUEUES` property (see `ecp.properties` File). If the pool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNPOOL WEIGHTQUEUES` DB field (see `HPSA_EQUIPMENTCONNPOOL` DB Table).

6.5.2.1. Pool Instance Specific Logging Parameters Configuration

LogLevel: The pool logger level. Should be an integer value. If the log message level value is greater or equal than the log level, the message will be written. The numerical values of the log4j log levels are:

```
FATAL = 50000
ERROR = 40000
WARN = 30000
INFO = 20000
DEBUG = 10000
ALL = Integer.MIN_VALUE
```

If the pool is dynamic, the value of this parameter will be specified by the `ecp.properties` `DYNAMIC_POOL_LOG_LEVEL` property (see `ecp.properties` File). If the pool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNPOOL LOGLEVEL` DB field (see `HPSA_EQUIPMENTCONNPOOL` DB Table).

Notice that each **SubPool** will use the logger of the pool it belongs to to log its messages (see **SubPool Instance Specific Logging Parameters Configuration**)

LogFilePath: Determines the file where the pool instance log will be written. The file will be created in the directory specified by the `ecp.properties` `LOG_DIR` property.

If the pool is dynamic, the **LogFilePath** will be

```
LOG_DIR + Pool.Name + ".log"
```

If the pool is static, the **LogFilePath** will be specified by the `HPSA_EQUIPMENTCONNPOOL LOGFILE` field (see `HPSA_EQUIPMENTCONNPOOL` DB Table).

```
LOG_DIR + LOGFILE
```

Notice that each SubPool will use the logger of the pool it belongs to to log its messages (see SubPool Instance Specific Logging Parameters Configuration)

6.5.2.2. Pool Instance Specific Clustering Parameters Configuration

Clustered: Whether the pool should be clustered. When ECP operates in cluster mode only clustered pools are loaded. Likewise, when ECP operates in non-cluster/legacy mode, only non-clustered pools are loaded. When **Clustered** is enabled, parameters **InstanceNames** and **NumInstances** are mandatory.

InstanceNames: Ordered comma-separated list of the names of the ECP instances that should load the pool. This parameter is mandatory for clustered pools.

NumInstances: The number of ECP instances that should simultaneously load the pool. The pool will be loaded in the **N** first instances of the **InstanceNames** list.

6.6. SubPool Configuration

6.6.1. SubPool Instance Specific Parameters Configuration

InitSessions: Determines whether the resources should be initialized as soon as created (or reused) or the SubPool should wait until the resource is requested by the client.

If the subpool is dynamic, and the advanced dynamic properties are set (see DynamicECPPProperties Advanced Properties) the value of this parameter will be specified by the DynamicECPPProperties **InitSessions** attribute (see DynamicECPPProperties Advanced SubPool Properties). If the advanced dynamic properties are not set, the value of this parameter will be specified by the `ecp.properties` **DYNAMIC_POOL_INIT_SESSIONS** property (see `ecp.properties` File). If the pool is static, the value of this parameter will be specified by the **HPSA_EQUIPMENTCONNSUBPOOL_INITSESSIONS** DB field (see **HPSA_EQUIPMENTCONNSUBPOOL** DB Table).

MaxSessions: Maximum number of resources that the SubPool will contain. Resources will be allocated as needed, but existent resources will be reused if possible.

If the subpool is dynamic, and the advanced dynamic properties are set (see DynamicECPPProperties Advanced Properties) the value of this parameter will be specified by the DynamicECPPProperties **InitSessions** attribute (see DynamicECPPProperties Advanced SubPool Properties). If the advanced dynamic properties are not set, the value of this parameter will be specified by the `ecp.properties` **DYNAMIC_POOL_MAX_SESSIONS** property (see `ecp.properties` File). If the pool is static, the value of this parameter will be specified by the **HPSA_EQUIPMENTCONNSUBPOOL_MAXSESSIONS** DB field (see **HPSA_EQUIPMENTCONNSUBPOOL** DB Table).

MinSessions: Minimum number of resources to keep instantiated in the SubPool. The SubPool will always contain at least that quantity of resources.

If the subpool is dynamic, and the advanced dynamic properties are set (see DynamicECPPProperties Advanced Properties) the value of this parameter will be specified by the DynamicECPPProperties **InitSessions** attribute (see DynamicECPPProperties Advanced SubPool Properties). If the advanced dynamic properties are not set, the value of this parameter will be specified by the `ecp.properties` **DYNAMIC_POOL_MIN_SESSIONS** property (see `ecp.properties` File). If the pool is static, the value of this parameter will be specified by the **HPSA_EQUIPMENTCONNSUBPOOL_MINSESSIONS** DB field (see **HPSA_EQUIPMENTCONNSUBPOOL** DB Table).

ResourceTimeout: Maximum time that a resource is allowed to remain BUSY (in use by a client). Once the timeout has expired the resource remains BUSY, the resource will be finalized (and eventually reinitialized and reassigned to other client).

If the subpool is dynamic, and the advanced dynamic properties are set (see `DynamicECPPProperties Advanced Properties`) the value of this parameter will be specified by the `DynamicECPPProperties InitSessions` attribute (see `DynamicECPPProperties Advanced SubPool Properties`). If the advanced dynamic properties are not set, the value of this parameter will be specified by the `ecp.properties DYNAMIC_POOL_RESOURCE_TIME_OUT` property (see `ecp.properties File`). If the pool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL RESOURCETIMEOUT` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL DB Table`).

TemporaryResourcesTimeout: Maximum time that a temporary connection can remain unused after it is created, in milliseconds. If the timeout is set to 0, it will never expire. Once the timeout has expired the temporary connection will be finalized and destroyed. Temporary connections are the additional connections to `MinSessions`. Expired temporary connections are not reused. Instead, they are finalized and destroyed after their expiration. They may be reused though, if the connection is not expired and the SubPool is reloaded or the connection closed (via RMI) and the pool contains less than `MinSessions` resources. Notice that Temporary connections are also affected by `ResourceTimeout`.

If the subpool is dynamic, and the advanced dynamic properties are set (see `DynamicECPPProperties Advanced Properties`) the value of this parameter will be specified by the `DynamicECPPProperties InitSessions` attribute (see `DynamicECPPProperties Advanced SubPool Properties`). If the advanced dynamic properties are not set, the value of this parameter will be specified by the `ecp.properties DYNAMIC_POOL_TEMPORARY_RESOURCES_TIME_OUT` property (see `ecp.properties File`). If the pool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL TEMPORARYRESOURCESTIMEOUT` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL DB Table`).

Id: Identifier of the subpool.

If the subpool is dynamic, then the subpool identifier will be 0 (actually, the JVM initialization value of an integer, as the value is not initialized by the ECP). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL IDSUBPOOL` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL DB Table`).

ConnectionResourceClassName: Fully qualified class name of the `EquipmentDriver` to be used for this subpool connections. Must extend `ConnectionResource` and be in the system codebase (classpath).

If the subpool is dynamic, the value of this parameter will be specified by the `DynamicECPPProperties ConnectionResourceClassName` attribute (see

`DynamicECPPProperties SubPool Properties`). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL CONNECTIONRESOURCECLASSNAME` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL DB Table`).

6.6.1.1. SubPool Instance Specific Logging Parameters Configuration

Logger: Each SubPool will use the logger of the pool it belongs to to log its messages (see `Pool Logging Common Parameters Configuration` and `Pool Instance Specific Logging Parameters Configuration`).

Notice that each `ConnectionResource` will use the logger of the SubPool it belongs to to log its messages (see `ConnectionResource Configuration`)

6.6.1.2. EquipmentDriver Initialization Parameters Configuration

Ip: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.host`. If the SubPool to which the EquipmentDriver belongs is dynamic, the value of this parameter will be specified by the `DynamicECPPProperties IP` attribute (see

`DynamicECPPProperties SubPool Properties`). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL IP` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL` DB Table).

Port: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.port`. If the SubPool to which the EquipmentDriver belongs is dynamic, the value of this parameter will be specified by the `DynamicECPPProperties Port` attribute (see

`DynamicECPPProperties SubPool Properties`). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL PORT` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL` DB Table).

Protocol: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.protocol`. If the SubPool to which the EquipmentDriver belongs is dynamic, the value of this parameter will be specified by the `DynamicECPPProperties Protocol` attribute (see

`DynamicECPPProperties SubPool Properties`). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL PROTOCOL` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL` DB Table). The value identifies a ProtocolDriver. A ProtocolDriver registered in the Protocol Driver manager under that name must exist. See Protocol Drivers Manager Configuration

Username: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.user`. If the SubPool to which the EquipmentDriver belongs is dynamic, the value of this parameter will be specified by the `DynamicECPPProperties User` attribute (see

`DynamicECPPProperties SubPool Properties`). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL USERNAME` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL` DB Table).

Password: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.password`. If the SubPool to which the EquipmentDriver belongs is dynamic, the value of this parameter will be specified by the `DynamicECPPProperties Password` attribute (see

`DynamicECPPProperties SubPool Properties`). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL PASSWORD` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL` DB Table).

Userenable: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.userEnable`. If the SubPool to which the EquipmentDriver belongs is dynamic, the value of this parameter will be specified by the `DynamicECPPProperties UserEnable` attribute (see

DynamicECPProperties SubPool Properties). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL PASSWORDENABLE` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL` DB Table).

`Passwordenable`: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.passwordEnable`. If the SubPool to which the EquipmentDriver belongs is dynamic, the value of this parameter will be specified by the `DynamicECPProperties PasswordEnable` attribute (see

DynamicECPProperties SubPool Properties). If the subpool is static, the value of this parameter will be specified by the `HPSA_EQUIPMENTCONNSUBPOOL PASSWORDENABLE` DB field (see `HPSA_EQUIPMENTCONNSUBPOOL` DB Table).

`ExternalDataGetterClassName`: Fully qualified class name of the external data getter implementation to be used for this subpool connections, which must extend `ExternalDataGetter` and be available in the classpath. This is an optional parameter, and in case it is present, an instance of the specified implementation will be created to retrieve connection information for the subpool based on the network element name. This avoids the need to explicitly configure subpool connection parameters such as the target equipment host, password, port, protocol, etc.

`ElementName`: Passed from the SubPool to the EquipmentDriver on construction in the Map entry key `ConnectionResource.DefaultParameterNames.elementName`. Network element name. This optional parameter is used in conjunction with the previous one to identify the name of the network element a subpool should connect to.

6.7. EquipmenDriver Configuration

`Logger`: This logger is set by the `ConnectionResource` (see `ConnectionResource Configuration`) and it will determine the `PrintWriter` and the initial `LogLevel`. `EquipmentDriver` won't use it for logging (except to log some error messages). Instead, it will use `PrintWriter`.

`PrintWriter`: Will be determined by the `Logger`. If the `Logger` has a `WritableFileAppender`, it will be used as `PrinterWriter`. In other case, `System.out` will be used.

`LogLevel`: Will determine the level of the messages to `PrintWriter` to print. Initially, `LogLevel` will be the `LogLevel` of `Logger` but the client can change it when requesting a connection. In fact, `CLICommands` will set the `EquipmentDriver LogLevel` to its own `LogLevel` when executing an `Operation` (see `CLICommands Configuration`).

if `LogLevel` equals `org.apache.log4j.Level.INFO` the `EquipmentDriver` will write on this `PrintWriter` the data read and written through the `ProtocolDriver`.

If `LogLevel` equals `org.apache.log4j.Level.DEBUG`, only the data read during `configureTerminal` will be logged but accumulating it, that is, if five consecutive read operations are needed to find a searched string, the five reading operations, each one including the previous read data, will be logged.

If an error is found, the read data will always be written.

6.7.1. EquipmentDriver Initialization Parameters Configuration

See `EquipmentDriver Initialization Parameters Configuration`

6.7.2. ConnectionResource Configuration

Logger: Each ConnectionResource will use the logger of the SubPool it belongs to, to log its messages (see SubPool Instance Specific Logging Parameters Configuration). It will also determine the Logger of the EquipmentDriver (see EquipmentDriver Configuration).

6.8. Protocol Drivers Manager Configuration

The Protocol Drivers Manager can be configured through the `ProtocolDrivers.lst` file. See `ProtocolDrivers.lst` File.

Additionally, the Protocol Drivers Manager will use the current `log4j` `LoggerRepository` to look for the logger with the name "DriverManager". A `ConsoleAppender` to `stdout` will be added to this logger, and used to log the Protocol Driver registering process.

6.9. ProtocolDriver Configuration

SpyFile: The directory where the spy files will be generated may be configured from the `ecp.properties` file `LOG_DIR` property. The spy file will be

```
LOG_DIR + "spy" + ${pool.name} + "_" + ${subpool.name} + "_" + ${resource.id} + ".log"
```

See `ecp.properties` File.

6.10. CLICommands Configuration

Logger: Default `AbstractLoggeable` Logger (`ConsoleAppender` to `System.out`):

LogLevel: On `CLICommands` construction, the Logger log level will be established to `org.apache.log4j.Level.INFO` if `bInfo==true`, or to `org.apache.log4j.Level.WARN` if `bInfo==false`. Later, this logger level will be used to determine the EquipmentDriver LogLevel, setting it to the same log level (see EquipmentDriver Configuration).

ECPRMIServiceRegistryHostName: Hostname of the registry service where the ECP RMI Service object has been bound. This parameter may be configured via `setRMIServiceName`.

ECPRMIServiceRegistryPort: Port of the registry service where the ECP RMI Service object has been bound. This parameter may be configured via `setRMIServiceName`.

ECPRMIServiceReferenceName: Name to which the ECP RMI Service reference has been bound. This parameter may be configured via `setRMIServiceName`.

6.11. Template Parser Configuration

`TemplateParser` logs to `System.out` but when constructing the `TemplateParser`, the generated `CLICommands` LogLevel can be set. The `bInfo` parameter will be passed to the `CLICommands`. See `CLICommands` Configuration.

6.12. JMS Monitoring Configuration

Enabling: To enable JMS Monitoring, the `ecp.properties` field `ECP.Msgs.Enable` must be set to `true`. No Monitoring message will be sent if this property is set to other value. See 0 `ecp.properties` File, property `"ECP.Msgs.Enable"` for further details.

Administered Objects Naming Context: All the properties included in `ecp.properties` will be set as environment of the `InitialContext` instance used to look for administered objects. By default, the `ecp.properties` will use the Active MQ Initial Context Factory, setting the property

`"java.naming.factory.initial=`

`org.apache.activemq.jndi.ActiveMQInitialContextFactory"`. Check Active MQ

documentation and `javax.naming.InitialContext` for the possible values. It is possible for example to create administered Destinations by tweaking these properties. See 0 `ecp.properties` File, property

`"java.naming.factory.initial"` for further details.

JMS Broker Connection/Start: It is possible to determine the JMS Broker that the ECP will connect to, indicating its URI in the `ecp.properties` field `JMSBrokerReference.broker.uri`. This URI will be used for to instantiate an `ActiveMQConnectionFactory`, to create connections to the broker. Active MQ supports a wide variety of URIs, including embedded brokers, broker configuration through URI, multiple transport protocols etc. Check Active MQ documentation for details. By default the JMS Broker URI is

`"vm\:(broker\:(tcp\://localhost\:4001)?brokerName\=EmbeddedBroker&useJmx\=true&persistent\=false&populateJMSXUserID\=false&useShutdownHook\=false&deleteAllMessagesOnStartup\=false&enableStatistics\=false)?marshal\=false"`. "vm" URIs will start an embedded broker. See 0 `ecp.properties` File, property

`"JMSBrokerReference.broker.uri"` for further details.

JMS ECP Messages Destination: It is possible to determine the destination type and name where ECP monitoring messages will be sent. See 0 `ecp.properties` File, properties

`"JMSMessageBroker.dest.type"` and `"JMSMessageBroker.dest.name"` for further details.