TWORKS/DFP 653.002

AutoPilot® TransactionWorks® Direct Feed Probe

Version 6.5.3

Installation and User's Guide

CONFIDENTIALITY STATEMENT: THE INFORMATION WITHIN THIS MEDIA IS PROPRIETARY IN NATURE AND IS THE SOLE PROPERTY OF NASTEL TECHNOLOGIES, INC. ALL PRODUCTS AND INFORMATION DEVELOPED BY NASTEL ARE INTENDED FOR LIMITED DISTRIBUTION TO AUTHORIZED NASTEL EMPLOYEES, LICENSED CLIENTS, AND AUTHORIZED USERS. THIS INFORMATION (INCLUDING SOFTWARE, ELECTRONIC AND PRINTED MEDIA) IS NOT TO BE COPIED OR DISTRIBUTED IN ANY FORM WITHOUT THE EXPRESSED WRITTEN PERMISSION FROM NASTEL TECHNOLOGIES, INC.

© 1998-2017 Nastel Technologies, Inc. All rights reserved.

PUBLISHED BY:

RESEARCH & DEVELOPMENT

NASTEL TECHNOLOGIES, INC.

88 SUNNYSIDE BLVD, SUITE 101

PLAINVIEW, NY 11803

COPYRIGHT © 1998-2017. ALL RIGHTS RESERVED. NO PART OF THE CONTENTS OF THIS DOCUMENT MAY BE PRODUCED OR TRANSMITTED IN ANY FORM, OR BY ANY MEANS WITHOUT THE WRITTEN PERMISSION OF NASTEL TECHNOLOGIES.

DOCUMENT TITLE: **NASTEL AUTOPILOT TRANSACTIONWORKS™ DIRECT FEED PROBE INSTALLATION AND USER'S GUIDE**

VERSION: **6.5.2**

DOCUMENT RELEASE DATE: **AUGUST 2017**

NASTEL DOCUMENT NUMBER: **TWORKS/DFP 653.002**

CONFIDENTIALITY STATEMENT: THE INFORMATION WITHIN THIS MEDIA IS PROPRIETARY IN NATURE AND IS THE SOLE PROPERTY OF NASTEL TECHNOLOGIES, INC. ALL PRODUCTS AND INFORMATION DEVELOPED BY NASTEL ARE INTENDED FOR LIMITED DISTRIBUTION TO AUTHORIZED NASTEL EMPLOYEES, LICENSED CLIENTS, AND AUTHORIZED USERS. THIS INFORMATION (INCLUDING SOFTWARE, ELECTRONIC AND PRINTED MEDIA) IS NOT TO BE COPIED OR DISTRIBUTED IN ANY FORM WITHOUT THE EXPRESSED WRITTEN PERMISSION FROM NASTEL TECHNOLOGIES, INC.

ACKNOWLEDGEMENTS:

THE FOLLOWING TERMS ARE TRADEMARKS OF NASTEL TECHNOLOGIES CORPORATION IN THE UNITED STATES OR OTHER COUNTRIES OR BOTH: TRANSACTIONWORKS, M6 AUTOPILOT, AUTOPILOT/IT, AUTOPILOT/ENTERPRISE, M6 FOR WMQ, AUTOPILOT/WMQ, M6 WEB SERVER, M6 WEB CONSOLE, AUTOPILOT/WEB, MQCONTROL, MQCONTROL EXPRESS, AUTOPILOT/TRANSACTION ANALYZER, AUTOPILOT/WAS, AUTOPILOT/TRANSACTION MONITOR, AUTOPILOT/OS MONITOR.

THE FOLLOWING TERMS ARE TRADEMARKS OF THE IBM CORPORATION IN THE UNITED STATES OR OTHER COUNTRIES OR BOTH: IBM, MQ, MQSERIES, WEBSPHERE, WEBSPHERE MQ WIN-OS/2, AS/400, OS/2, DB2, AND AIX, Z/OS.

THE FOLLOWING TERMS ARE TRADEMARKS OF HEWLETT-PACKARD IN THE UNITED STATES OR OTHER COUNTRIES OR BOTH: OPENVIEW, HP-UX.

COMPAQ, THE COMPAQ LOGO, ALPHASERVER, COMPAQ INSIGHT MANAGER, CDA, DEC, DECNET, TRUCLUSTER, ULTRIX, AND VAX REGISTERED IN U.S. PATENT AND TRADEMARK OFFICE. ALPHA AND TRU64 ARE TRADEMARKS OF COMPAQ INFORMATION TECHNOLOGIES GROUP, L.P IN THE UNITED STATES AND OTHER COUNTRIES.

SNMPC, SNMPC, WORKGROUP, AND SNMPC ENTERPRISE ARE TRADEMARKS OF CASTLE ROCK COMPUTING IN THE UNITED STATES OR OTHER COUNTRIES, OR BOTH.

SUN, SUN MICROSYSTEMS, THE SUN LOGO, IFORCE, JAVA, NETRA, N1, SOLARIS, SUN FIRE, SUN RAY, SUNSPECTRUM, SUN STOREDGE, SUNTONE, THE NETWORK IS THE COMPUTER, ALL TRADEMARKS AND LOGOS THAT CONTAIN SUN, SOLARIS, OR JAVA, AND CERTAIN OTHER TRADEMARKS AND LOGOS ARE TRADEMARKS OR REGISTERED TRADEMARKS OF ORACLE CORPORATION AND/OR ITS AFFILIATES.

INSTALLANYWHERE IS A REGISTERED TRADEMARK OF ZEROG SOFTWARE IN THE UNITED STATES OR OTHER COUNTRIES, OR BOTH.

THIS PRODUCT INCLUDES SOFTWARE DEVELOPED BY THE APACHE SOFTWARE FOUNDATION (HTTP://WWW.APACHE.ORG/). THE JAKARTA PROJECT" AND "TOMCAT" AND THE ASSOCIATED LOGOS ARE REGISTERED TRADEMARKS OF THE APACHE SOFTWARE FOUNDATION

INTEL, PENTIUM AND INTEL486 ARE TRADEMARKS OR REGISTERED TRADEMARKS OF INTEL CORPORATION IN THE UNITED STATES, OR OTHER COUNTRIES, OR BOTH

MICROSOFT, WINDOWS, WINDOWS NT, WINDOWS XP, .NET, .NET FRAMEWORK AND THE WINDOWS LOGOS ARE REGISTERED TRADEMARKS OF THE MICROSOFT CORPORATION.

UNIX IS A REGISTERED TRADEMARK IN THE UNITED STATES AND OTHER COUNTRIES LICENSED EXCLUSIVELY THROUGH X/OPEN COMPANY LIMITED.

"LINUX" AND THE LINUX LOGOS ARE REGISTERED TRADEMARKS OF LINUS TORVALDS, THE ORIGINAL AUTHOR OF THE LINUX KERNEL. ALL OTHER TITLES, APPLICATIONS, PRODUCTS, AND SO FORTH ARE COPYRIGHTED AND/OR TRADEMARKED BY THEIR RESPECTIVE AUTHORS.

SCO CUSA, SCO DOCTOR, SCO DOCTOR FOR NETWORKS, SCO DOCTOR LITE, SCO GLOBAL ACCESS, SCO MPX, SCO MULTIVIEW, SCO NIHONGO OPENSERVER, SCO OK, THE SCO OK LOGO, SCO OPENSERVER, SCO OPEN SERVER, SCO PORTFOLIO, SCO POS SYSTEM, SCO TOOLWARE, AND THE WORLD NEVER STOPS ARE TRADEMARKS OR REGISTERED TRADEMARKS OF CALDERA INTERNATIONAL, INC. IN THE U.S.A. AND OTHER COUNTRIES, ALL RIGHTS RESERVED.

ORACLE® IS A REGISTERED TRADEMARK OF ORACLE CORPORATION AND/OR ITS AFFILIATES

OTHER COMPANY, PRODUCT, AND SERVICE NAMES, MAY BE TRADEMARKS OR SERVICE MARKS OF OTHERS.

Contents

CHAPTER 1: INTRODUCTION 1

1.1 How This Guide is Organized 1
1.2 History of this Document 1
1.3 Related Documents 2
1.4 Release Notes 2
1.5 Intended Audience 2
1.5.1 User Feedback 2
1.6 Technical Support 2
1.7 Terms and Abbreviations 2
1.8 Conventions 2

CHAPTER 2: ABOUT AUTOPILOT TRANSACTIONWORKS DIRECT FEED PROBE 3

2.1 AutoPilot TransactionWorks Architecture 3
2.2 AutoPilot TransactionWorks Direct Feed Probe 4
2.3 AutoPilot TransactionWorks Definitions 6

CHAPTER 3: AUTOPILOT TRANSACTIONWORKS DIRECT FEED PROBE INSTALLATION 7

3.1 Before Installation 7
3.1.1 Technical Documents 7
3.1.2 Installation Requirements 7
3.1.3 Contents of AutoPilot TransactionWorks Distribution 7
3.2 Installing AutoPilot TransactionWorks Direct Feed Probe 8

CHAPTER 4: CONFIGURING AUTOPILOT TRANSACTIONWORKS DIRECT FEED PROBE 9

4.1 Properties 9
4.2 Configuring Parsers 10
4.2.1 Configuring Parser Fields 11
4.3 Configuring Feeders 15

CHAPTER 5: USING BUILT-IN COMPONENTS 17

5.1 Built-in Parsers 17
5.1.1 Tokenizer Parser 17
5.1.2 Name/Value Pair Parser 18
5.1.3 Regular Expression Parser 18
5.1.4 XML Parser 19
5.2 Built-in Feeders 19
5.2.1 Line-Oriented File Feeder 20
5.2.2 Stream Feeder 20
5.2.3 WMQ Feeder 20

CHAPTER 6: APPLYING DIRECT FEED PROBE 21

6.1 Log File Analysis 21
6.1.1 Simple Log File 21
6.1.2 Multiple Log Files 23
6.2 Transaction Stitching 23

CHAPTER 7: EXECUTING DIRECT FEED PROBE 25

 © 1998-2017 Nastel Technologies, Inc.

CHAPTER 8: BUILDING CUSTOM COMPONENTS 27

8.1 BUILDING CUSTOM PARSERS 27
8.2 BUILDING CUSTOM FEEDERS 27
8.3 OTHER RECOMMENDATIONS 28
8.4 INTEGRATING DIRECT FEED PROBE INTO APPLICATIONS 29
8.4.1 Integration with Log4j 29

APPENDIX A: REFERENCES 31

A.1 NASTEL DOCUMENTATION 31
A.2 DOCUMENTATION 31

APPENDIX B: CONVENTIONS 33

B.1 TYPOGRAPHICAL CONVENTIONS 33

GLOSSARY 35

INDEX 39

Figures

FIGURE 2-1. AUTOPILOT M6/TRANSACTIONWORKS ARCHITECTURE 5

Tables

TABLE 1-1. DOCUMENT HISTORY 1
TABLE A-1. NASTEL DOCUMENTATION 31
TABLE B-1. TYPOGRAPHICAL CONVENTIONS 33

This page intentionally left blank.

Chapter 1: Introduction

Welcome to the *Nastel AutoPilot TransactionWorks Direct Feed Probe Installation and User's Guide*. This guide describes installation, configuration, and user interface. Please review this guide carefully before installing and using the product.

1.1 How This Guide is Organized

Chapter 1: Identifies the users and history of the document as well as supplying support and reference information.

Chapter 2: A brief overview of AutoPilot TransactionWorks Direct Feed Probe and its components.

Chapter 3: Describes AutoPilot TransactionWorks Direct Feed Probe's automatic and manual installation.

Chapter 4: Describes AutoPilot TransactionWorks Direct Feed Probe's configuration.

Chapter 5: Provides sample applications.

Chapter 6: Describers how the Direct Feed Probe can be applied in a variety of situations.

Chapter 7: Describes executing Direct Feed Probe.

Chapter 8: Describes the requirements for defining custom parsers and/or custom feeders.

Appendix A: Provides a list of reference information for using AutoPilot TransactionWorks Probes.

Appendix B: Contains conventions used in this document.

Glossary: Contains a listing of unique and common acronyms and words and their definition.

Index: Contains an alphanumeric cross-reference of all topics and subjects of importance.

1.2 History of this Document

Table 1-1. Document History

Release Date	Document Number	AutoPilot TransactionWorks Version	Summary
September 2011	TWORKS/DFP 600.001	6.0.18	Initial release
April 2013	TWORKS/DFP 650.001	6.5	Update for Version 6.5
May 2013	TWORKS/DFP 650.002	6.5	Errata
September 2013	TWORKS/DFP 650.003	6.5	Added additional properties
February 2014	TWORKS/DFP 650.004	6.5.2	Update for Version 6.5.2
December 2015	TWORKS/DFP 653.001	6.5.3	Update for Version 6.5.3 (Mantis 9706, 11580)
August 2017	TWORKS/DFP 653.002	6.5.3	Errata XML parser (Mantis 11580), update Nastel's phone numbers and street address

 © 1998-2017 Nastel Technologies, Inc.

1.3 Related Documents

A complete listing of related and referenced documents is in Appendix A of this guide.

1.4 Release Notes

See the **README.htm** files on your installation media or AutoPilot TransactionWorks installation directory. Release notes and updates are also available through the Nastel Resource Center at: http://www.nastel.com/resources.

1.5 Intended Audience

This document is intended for personnel installing, configuring, and using AutoPilot TransactionWorks Probes. The person installing and configuring AutoPilot TransactionWorks Probes should be familiar with:

- Java 1.6 and higher

1.5.1 User Feedback

Nastel encourages all users of AutoPilot TransactionWorks to submit comments, suggestions, corrections, and recommendations for improvement for all AutoPilot TransactionWorks documentation. Please send your comments via mail or e-mail. Send e-mail messages to: support@nastel.com. You will receive a written response, along with status of any proposed change, update, or correction.

1.6 Technical Support

If you need additional technical support, you can contact Nastel by telephone or by e-mail.

- To contact Nastel technical support by telephone, call **800-963-9822 ext. 1**. If you are calling from outside the United States, dial **001-516-801-2100**.
- To contact Nastel technical support by e-mail, send a message to support@nastel.com.
- To contact Nastel technical support through the support website (user ID and password are required), go to http://support.nastel.com/btracker, or visit the Nastel Resource Center at: http://www.nastel.com/resources.

Contact your local AutoPilot TransactionWorks administrator for further information.

1.7 Terms and Abbreviations

A list of terms and abbreviations used in all AutoPilot TransactionWorks document is located in the Glossary.

1.8 Conventions

Refer to Appendix B for typographical and naming conventions used in all AutoPilot TransactionWorks documentation.

Chapter 2: About AutoPilot TransactionWorks Direct Feed Probe

AutoPilot TransactionWorks Direct Feed Probe is a *lightweight performance measurement and monitoring* tool for live applications. It records transaction related data, response time, and exceptions for activities initiated from applications and Web sessions. This tool can be used to detect bottlenecks and failures. *AutoPilot TransactionWorks Direct Feed Probe is designed for live applications in development, QA, and production environments, with minimal performance overhead*.

The **AutoPilot TransactionWorks Direct Feed Probe** allows the user to feed data from a variety of sources to the TransactionWorks Analyzer. Some examples of sources are:

- Log files
- Data streams (including sockets)

The probe can be run either as a standalone server application or can be integrated into existing applications.

2.1 AutoPilot TransactionWorks Architecture

AutoPilot TransactionWorks (Figure 2-1) has three major components:

- Transaction Probes (TPs)
- Transaction Analyzer (TA)
- AutoPilot TransactionWorks Explorer

TPs are application programs that are executed at a user-defined place in transaction processing applications and collect transaction-related data. TPs intercept the transaction message data and publish it to the TA.

TA is an AutoPilot M6 expert (can also be executed as a Web Service or Standalone application) that collects the transaction message data (published by TPs), analyzes it in real-time, and publishes metric facts to the AutoPilot M6 facts board as summary statistics. The facts collected can then be recorded in a database for report generation, and/or used to build business views that address your specific needs.

TA has two transport options:

- TCP/IP (Java, .NET, WMQ Probe, and Direct Feed)
- HTTP (Java and Direct Feed)

  JMS is only available when running TA as a Web Service. TA can be configured to collect transaction details via either one, or all of the transports simultaneously depending on the transaction probes configuration. It can also log transaction details to any database via Java Database Connectivity (JDBC) configuration.

The **Explorer** provides a graphical view of transaction processing network metrics.

 © 1998-2017 Nastel Technologies, Inc.

2.2 AutoPilot TransactionWorks Direct Feed Probe

AutoPilot TransactionWorks Direct Feed Probe allows data collected by other, possibly remote, applications to be fed directly to TransactionWorks Analyzer, applying the necessary conversions to translate the information into the format recognizable by the TransactionWorks Analyzer. For example, a log file containing a series of operation events can be converted to represent operations in a "transaction", which allows grouping of related events. The probe can either be run as a standalone application, or integrated into an existing application.

The TransactionWorks Direct Feed Probe consists of the following components:

- **Activities** – represent some action to be recorded in the TransactionWorks Analyzer. Some examples of an activity include operations, events, etc.
- **Parsers** – process the raw activity data from a feeder, extracting the activity fields from the raw data and mapping them to the appropriate activity property.
- **Feeders** – extract the raw activity data from a source. Some examples of feeder sources include files, streams, queues, etc.
- **Configuration** – defines the parser and feeder instances.

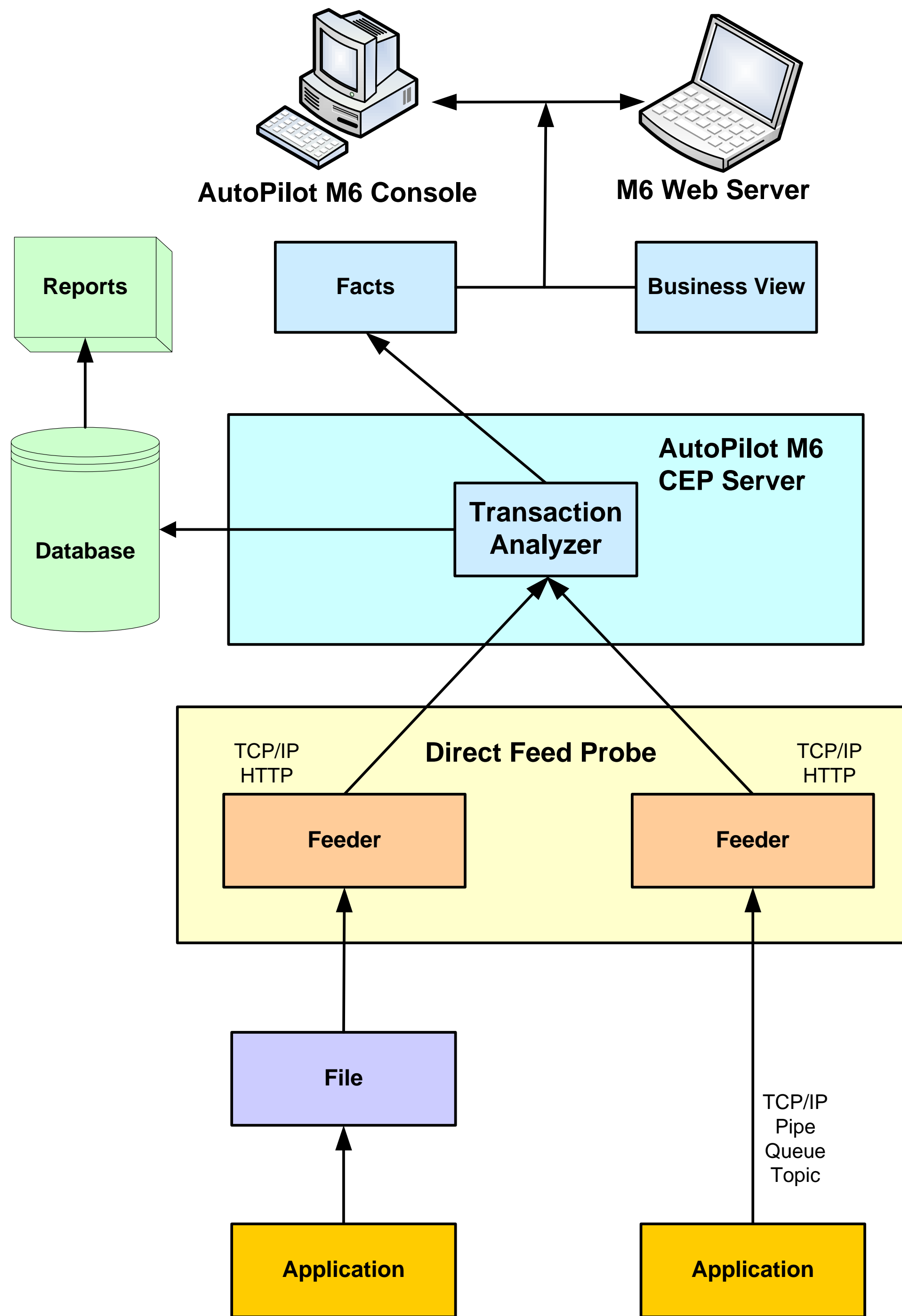

Figure 2-1. AutoPilot M6/TransactionWorks Architecture

© 1998-2017 Nastel Technologies, Inc.

2.3 AutoPilot TransactionWorks Definitions

AutoPilot TransactionWorks defines the following concepts as follows:

Activity – Represents a collection of events that should be considered to be a single application activity.

Application – Represents a logical collection of software components that perform a business function running on a specific server.

Duration – The "clock" time that a transaction took to complete. This is the difference between the time of the first send operation and the end of the last LUW.

Event – Represents a specific action taken by an application.

Logical Unit of Work (LUW) – Represents a collection of operations and messages within a session that should be considered to be a single unit of work (all or nothing property). These are generally delimited by START/COMMIT calls.

Message – Represents a physical message being transported through the network.

Message Age – The time that a message was in a resource waiting to be processed.

Operation – Represents a specific operation, such as send, receive, connect, commit, etc.

Resource – An entity on which transactions are executed or a medium of exchange (e.g., Queue, DB table, file, JMS topic, etc).

Resource Manager – An entity that is managing a collection of resources. Examples include a WMQ Queue Manager, Application Server, and Database Server.

Runtime – The period during which the program is executing.

Session – Represents a specific period of execution of an application. Examples include the interval during which a database or queue manager connection is active.

Transaction – A group of activities targeted at achieving a common goal or a task. It is represented by a collection of related LUWs. This relationship is determined based on the messages exchanged between the LUWs.

Wait Time – The time that a receive operation was blocked waiting for a message to arrive.

Workload – The sum total of the execution times of all LUWs in a transaction.

Chapter 3: AutoPilot TransactionWorks Direct Feed Probe Installation

This chapter provides instructions for setup requirements and a typical AutoPilot TransactionWorks installation. Installation can be performed using either pre-built scripts or manual procedures. The steps and procedures covered in this chapter are described in the following sections:

- 3.1 Before Installation
- 3.2 Installing AutoPilot TransactionWorks Direct Feed Probe

3.1 Before Installation

The procedures in this chapter cover the installation of the AutoPilot TransactionWorks Direct Feed Probe.

3.1.1 Technical Documents

Prior to installation you should review all text files and installation procedures. You should print, as needed, all of the installation-related materials to give yourself quick access to any required information during any installation procedures. Additional sets of printed documents are available from your Nastel representative or Nastel Support.

3.1.2 Installation Requirements

The requirements outlined in the paragraphs below specify the minimum requirements for AutoPilot TransactionWorks.

3.1.2.1 Hardware Requirements

- Hardware and operating system compatible with Java 1.6
- 512 MB RAM or higher

3.1.2.2 System Requirements

- Java Runtime Environment 1.6+
- WMQ Java libraries (Client install will suffice)

  IMPORTANT: For 64-bit installation 64-bit WMQ Java libraries require to chain before 32-bit WMQ Java libraries in the PATH environment variable in order for the Direct Feed probe application to connect to a WebSphere MQ queue manager using Java bindings.

3.1.3 Contents of AutoPilot TransactionWorks Distribution

The AutoPilot TransactionWorks distribution package consists of the following:

- books – AutoPilot TransactionWorks user documentation
- explorer – AutoPilot TransactionWorks Explorer files
- probes – AutoPilot TransactionWorks Probes files, organized by probe type
- analyzer – AutoPilot TransactionWorks Transaction Analyzer files
- samples – Sample applications for demonstrating AutoPilot TransactionWorks features.

 © 1998-2017 Nastel Technologies, Inc.

3.2 Installing AutoPilot TransactionWorks Direct Feed Probe

Extract ***<tworks_home>*/probes/direct-feed** directory from installation package to desired location. Before being able to run the probe, it needs to be configured, as explained in Chapter 4, Configuring AutoPilot TransactionWorks Direct Feed Probe. Once the configuration is complete, the probe is ready to be run.

Chapter 4: Configuring AutoPilot TransactionWorks Direct Feed Probe

There are two parts to configuring the Direct Feed Probe:

1. Defining the parser instances
2. Defining the feeder instances, associating them with the required parsers.

4.1 Properties

Properties are used to define the behavior of parsers and feeders. Properties are identified by name, and the set of properties supported by a parser or feeder, as well as how those properties are interpreted, are specific to that class of item. There are predefined properties that are supported by the built-in parsers and feeders. However, additional properties can be referenced by a parser definition (used with user-defined custom parsers). Parsers and feeders will ignore any properties they don't recognize. In addition, properties can be either required or optional, depending on the parser or feeder. If a required property is not defined, the initialization of the parser or feeder will fail. The built-in properties (along with their general interpretations) are:

- **`Channel`** – represents a communications channel
- **`DateTime`** – represents an initial, base, or default date, time, or date/time
- **`FieldDelim`** – represents the delimiter between fields in raw activity data
- **`FileName`** – represents a file name
- **`Host`** – represents a server host name or IP Address
- **`Namespace`** – represents the name of a namespace
- **`Pattern`** – a regular expression pattern
- **`Port`** – communications port number to either connect to or listen for connections on
- **`Queue`** – represents a queue name
- **`Queue Manager`** – represents a queue manager name
- **`RequireDefault`** – value is true or false
- **`SignatureDelim`** – string representing delimiter to use between components used to compute unique signature to identify the message associated with an activity
- **`StripHeaders`** – flag indicating whether these headers should be stripped from the data, for environments in which the data can have headers along with the user portion
- **`StripQuotes`** – flag indicating whether double quotes should be stripped from field value (expected to be set to "true" or "false")
- **`Subscription`** – represents the name that will be given to the subscription generated when subscribing to a topic using Publish/Subscribe framework. Used with **`Topic`** and/or **`TopicString`**.
- **`Topic`** – represents a topic object to subscribe when using Publish/Subscribe framework to connect to the tracking event stream
- **`TopicString`** – represents a fully qualified topic name or a topic set using wildcard characters string to subscribe when using Publish/Subscribe framework to connect to the tracking event stream
- **`ValueDelim`** – represents the delimiter between the label and value for a field in raw activity data

© 1998-2017 Nastel Technologies, Inc.

4.2 Configuring Parsers

The first step in configuring the Direct Feed Probe is to define the required parser instances. Here is a portion of a parser definition showing the various parts of the configuration:

```
<parser name="RegExParser"
        class="com.nastel.janus.probe.directfeed.parsers.ActivityRegExParser">
  <property name="Pattern" value="([-0-9]* +[:,\.0-9]*) +(.*) +(\[.*\])+ - (.*)"/>
  <field name="ApplName" value="TestApp"/>
  <field name="StatusCode" locator="2" datatype="String">
    <field-map source="INFO" target="SUCCEEDED"/>
    <field-map source="ERROR" target="FAILED"/>
  </field>
  <field name="Resource" separator="###">
    <field-locator locator="3" locator-type="REGroupNum"/>
    <field-locator locator="4" locator-type="REGroupNum">
      <field-map source="Starting..." target="TestMapping"/>
    </field-locator>
  </field>
</parser>
```

A parser definition contains the following attributes:

- **name** – unique name for the parser
- **class** – Java class implementing parser

A parser definition contains the following sub-elements:

- **<property>** – defines the setting for a property of the parser instance. The set of supported properties depends on the class of parser. See section 4.1, Properties.
- **<field>** – defines the transaction operation fields to set and how to extract the values for these fields from the raw activity data. Fields can optionally contain the following sub-elements:
    - **<field-map>** – defines how to translate values extracted from raw activity data to the required transaction operation values.
    - **<field-locator>** – defines the location or position in the raw activity data to extract the value from. A field can have multiple locators, which implies that the value for the field comes from multiple locations in the activity data. Each **<field-locator>** can have its own set of **<field-map>** entries.

4.2.1 Configuring Parser Fields

The field definitions indicate how to extract the value for each operation field from the raw activity data, and include the following attributes:

- **`name`** – name of activity field, which must be one of the supported fields. See list below.
- **`separator`** – string to insert between values when concatenating multiple values from raw activity data to form field value.
- Field definition also includes the following **`<field-locator>`** attributes:
  - **`locator`** – defines how to locate the value for the field. It can be comma-separated list of items, indicating the field value is the concatenation of the values at the specified locations, with each extracted value formatted using the same rules and separated by the string defined by **`separator`**. How this is interpreted depends on the class of parser, as well as the **`locator-type`** value (for parsers that support multiple **`locator`** types. Examples of locators supported by built-in parsers are:
    - position in a delimited string (e.g. CSV)
    - name for name/value pairs
    - regular expression group number
    - XPath expression
  - **`locator-type`** – indicates how to interpret the **`locator`**. Parsers that only support one type of **`locator`** will ignore this and assume **`locator`** is of the supported type. The recognized values for this attribute are parser-specific. The built-in ones are:
    - **`FeederProp`** – locator string is the name of a property supported by the parent feeder, causing the value of the field to be the value of the feeder property
    - **`Index`** – locator is a generic index, offset, or position within the raw activity data, e.g. field position in a CSV string
    - **`Label`** – locator is the name for a name/value set in the raw activity data
    - **`REGroupNum`** – locator is the regular expression group number to retrieve value from
    - **`REMatchNum`** – locator is the regular expression match iteration to retrieve value from
  - **`datatype`** – indicates how to interpret the value extracted from the raw activity data. This attribute is only relevant for fields whose value can be represented in several ways (and ignored for fields that are simple strings). An example of such a field is a date, which can either be a formatted string that needs to be parsed, a numeric timestamp. The supported data types are:
    - **`String`** – raw value is a character string (this is the default data type)
    - **`Binary`** – raw value is a sequence of bytes, represented in the specified **`format`**.
    - **`Number`** – raw value represents a numeric value
    - **`DateTime`** – raw value represents a date/time expression in the specified **`format`** (e.g. `2011-08-15 17:21:56.426385`)
    - **`Timestamp`** – raw value is a numeric timestamp in the specified **`units`**.
  - **`radix`** – indicates the radix (base) for a numeric value.
  - **`required`** – indicates if the field is required (true/false)
  - **`units`** – indicates how to interpret a numeric value. The supported units are parser- and field-specific. The current built-in unit types are:
    - **`Seconds`**
    - **`Milliseconds`** (default)
    - **`Microseconds`**

© 1998-2017 Nastel Technologies, Inc.

- **`format`** – indicates the format that the field value is in. The supported formats depend on the parser and field data type.
  - **`string`** – indicates that field is a character string. This is the general assumption if no format is specified.
  - For **`Binary`** data, **`format`** is one of:
    - **`base64Binary`** – raw value is the base64-encoded representation of a binary sequence
    - **`hexBinary`** – raw value is the hex-string representation of a binary sequence
  - For numeric data, **`format`** is a character string describing the format of the number. This is only required if value is not a standard representation of a real number. The supported formats are those supported by the class `java.text.DecimalFormat`. An example of a format string is "`#,##0.00`". Some of the more common format specifiers are:

0	Digit
#	Digit, with zero as absent
.	Decimal separator or monetary decimal separator
-	Minus sign
,	Grouping separator
E	Separates mantissa and exponent in scientific notation
%	Divide by 100 to convert from percentage

  - For **`DateTime`** data, **`format`** is a character string describing the format of the date/time expression. The supported formats are those supported by the class `java.text.SimpleDateFormat`, with the addition of support for microsecond fractional seconds (by using a format specification of "`SSSSSS`" for fractional seconds). An example of a format string is "`yyyy-MM-dd HH:mm:ss.SSSSSS`". Some of the more common format specifiers are:

y	Year (2- or 4-digit)
M	Month in year
d	Day in month
a	Am/pm marker
H	Hour in day (0-23)
k	Hour in day (1-24)
K	Hour in am/pm (0-11)
h	Hour in am/pm (1-12)
m	Minute in hour
s	Second in minute
S	Microseconds (if more than 3 specifiers), otherwise milliseconds
z	General Time zone (e.g. PST; GMT-08:00)
Z	RFC 822 Time zone (e.g. -0800)

    See documentation for `java.text.SimpleDateFormat` for a detailed description of the format specifiers.
- **`timezone`** – indicates the time zone that a date/time string is represented in. This is needed when the date/time string does not contain the time zone and the date/time string represents a date/time not in the local time zone. The time zone is specified as a string, as supported by `java.util.TimeZone.getTimeZone(String).`

  - **value** – indicates that the value for this field is the specified constant value, instead of deriving it from the raw activity data. Useful when the field is not represented in the raw activity data, but it's desired to record a value for this field, where this value is constant for all activity data.

Field definitions can also contain the following sub-elements:

- **<field-locator>** – defines how to locate the value in the raw activity data, as well as the format it is in. This element contains the attributes defined above. Multiple **< field-locator>** sub-elements can be specified to set the value for the field as the concatenation of multiple items from raw activity data, with each item separated by the **separator** string defined in the parent **<field>** element. The locator information for a field can be specified either using this sub-element, or via the properties directly in the **<field>** element, but not with both. The only time that this sub-element must be used is when each raw item must be interpreted and/or formatted differently.
- **<field-map>** – defines how to convert values in the raw activity data to its corresponding supported activity field value. This element has the following attributes:
  - **source** – raw activity data value to map. Specifying an empty source value (e.g., `source=""`) indicates that the target value for this entry should be used as the default value if raw activity data value does not match any of the source entries in this map.
  - **target** – value to map **source** to

  If there is no mapping defined for a raw activity data item (and no default value defined), then the value is used unchanged.

The set of supported fields is:

- **ServerName** – Host name of server to associate with activity
- **ServerIp** – IP Address of server to associate with activity
- **ServerOs** – String identifying information (e.g. type, version) about operating system associated with activity
- **ApplName** – Name of application associated with the activity. This field is required.
- **UserName** – Name of user associated with the activity
- **ResourceMgr** – Name of resource manager owning resource associated with the activity
- **ResMgrType** – Type of resource manager owning resource associated with the activity. Supported types are:
  - UNKNOWN
  - APP_SERVER
  - CICS_REGION
  - DATABASE_SERVER
  - MESSAGING_SERVER
  - STANDALONE_SERVER
- **Resource** – Name of resource associated with the activity

© 1998-2017 Nastel Technologies, Inc.

- **ResType** – Type of Resource associated with the activity. Supported types are:
  - UNKNOWN
  - C_RUNTIME
  - CORBA_OBJECT
  - DATABASE
  - DNET_RUNTIME
  - ENQ_MODEL
  - FILE
  - JAVA_RUNTIME
  - JMS_DESTINATION
  - JMS_QUEUE
  - JMS_TOPIC
  - JOURNAL
  - MAP
  - MSMQ_QUEUE
  - NETWORK
  - OS_RUNTIME
  - PARTITION
  - PORTAL_APP
  - PROGRAM
  - WEB_APP
  - WEB_SERVICE
  - WMQ_CHANNEL
  - WMQ_QMGR
  - WMQ_QUEUE
  - WMQ_TOPIC
- **ActivityName** – Name to assign to activity entry. Examples are operation, method, API call, event, etc. This is a required field.
- **ActivityType** – Type of activity. Supported types are:
  - OTHER
  - BROWSE
  - CALL
  - CLOSE
  - END
  - INQUIRE
  - OPEN
  - RECEIVE
  - SEND
  - SET
  - START
  - URL
- **StartTime** – Time action associated with activity started
- **EndTime** – Time action associated with activity ended
- **ElapsedTime** – Elapsed time of the activity
- **StatusCode** – Indicates completion status of the activity. Supported statuses are:
  - FAILED
  - SUCCEEDED
  - WARNING
- **ReasonCode** – Numeric reason/error code associated with the activity
- **ErrorMsg** – Error/exception message associated with the activity
- **Signature** – Identifier used to uniquely identify the data associated with this activity
- **Transport** – Message transport type where activity was observed. Supported values are:
  - UNKNOWN
  - DNET
  - FILE
  - HTTP
  - HTTPS
  - JMS
  - RMI
  - SMTP
  - SOAP
  - SQL
  - TCP
  - TIBCO
  - TIBCO_RV
  - UDP
  - WMQ
  - SSL
- **Tag** – User-defined label to associate with the activity, generally for locating activity
- **Correlator** – Identifier used to correlate/relate activity entries to group them into logical entities
- **ActivityData** – User data to associate with the activity
- **Value** – User-defined value associated with the activity (e.g. monetary value)

The following rules apply to processing fields for an activity entry:

- If a required field is not specified, then activity data is not recorded.
- Only one of **ServerName** or **ServerIp** need to be specified. The one that is not specified will be derived from the one that is. If neither are specified, the local system name and IP Address that feeder is running on is assumed.
- For **StartTime**, **EndTime**, and **ElapsedTime**, whichever ones are not specified are derived from the others, as follows:
  - If none specified, current time that event is being logged is used for both start and end times, with an elapsed time of 0
  - If both start and end times are specified, elapsed time is derived from difference between them
  - If only one of start or end times is specified, other is derived from specified one and the elapsed time (using 0 if not specified)
- If **StatusCode** is not specified, a status of `SUCCEEDED` is used.

4.3 Configuring Feeders

The next step in configuring the Direct Feed Probe is to define the required feeder instances. Here is a portion of a feeder definition showing the various parts of the configuration:

```
<feeder name="FileFeeder"
        class="com.nastel.janus.probe.directfeed.feeders.FileLineFeeder">
  <ta-conn protocol="TCP" host="localhost" port="6400"/>
  <property name="FileName" value="activity-data.log"/>
  <parser-ref name="RegExParser"/>
</feeder>
```

A feeder definition contains the following attributes:

- **name** – unique name for the feeder
- **class** – Java class implementing feeder

A feeder definition contains the following sub-elements:

- **<property>** – defines the setting for a property of the feeder instance. The set of supported properties depends on the class of feeder. See section 4.1, Properties.
- **<ta-conn>** – defines the connection information to use to connect to the TransactionWorks Analyzer. This element has the following properties:
  - **protocol** – communications protocol to use to connect to analyzer. Supported protocols are:
    - **FILE** – not really a communications protocol, but can be used to just record transaction activity to a file. Mainly useful for testing.
    - **HTTP** – use HTTP to connect and send data to analyzer
    - **HTTPS** – use HTTPS to connect and send data to analyzer
    - **TCP** – use TCP/IP to connect and send data to analyzer
  - **host** – host name or IP Address of server where analyzer is running. Not used with **FILE** protocol.
  - **port** – communications port that analyzer is listening on for connections. Not used with **FILE** protocol.
  - **file** – name of file to write transaction activity to. Only used with **FILE** protocol.

  - **`access-token`** – access token to use to authenticate connection to analyzer. Not used with **`FILE`** protocol.
  - **`proxy-host`** – host name or IP Address of proxy server to use to connect analyzer. Only used with HTTP and HTTPS protocols.
  - **`proxy-port`** – port number for proxy server to use to connect to analyzer. Only used with HTTP and HTTPS protocols.
  - **`keystore`** – path to keystore file containing analyzer's SSL certificate. Only used with HTTPS.
  - **`keystore-pwd`** – password for keystore. Only used with HTTPS.
- **`<parser-ref>`** – assigns the specified parser to this feeder. A feeder can have multiple parser references. The parsers are applied to each raw activity data item until a parser successfully processes the raw activity data. However, not every parser is compatible with every feeder. A parser handle data in one or more formats, and as a result, can only be used with feeders that can provide the raw activity data in that format. See Chapter 5, Using Built-in Components for a description of each of the included parsers and feeders.

Chapter 5: Using Built-in Components

The Direct Feed Probe contains several predefined parsers and feeders to handle some common sources and formats of activity data. For a detailed description of each of the parsers or feeders, see the reference document for the Direct Feed Probe.

5.1 Built-in Parsers

The Direct Feed Probe contains several predefined parsers. Each parser expects the data to be specified in a particular format (some support multiple formats). So feeders should only parsers that support the data format of the raw activity data being supplied by the feeder. The parsers will ignore data supplied in a format that is not supported. They also set the **`ActivityData`** field to the raw activity data being parsed. This can be overridden by providing a field definition for **`ActivityData`**.

5.1.1 Tokenizer Parser

The Tokenizer parser splits the raw activity data into a list of tokens, with each separated by the given delimiter(s). The locators for each field are 1-based numeric token positions. This parser is useful for parsing comma-delimited (CSV) strings.

The Tokenizer parser is implemented by the `com.nastel.janus.probe.directfeed.parsers.ActivityTokenParser` class, and supports the following properties:

- **`FieldDelim`** – String of characters containing the delimiters. Any character in this string is considered a delimiter. Default delimiter is "`,`" (comma).
- **`Pattern`** – Regular expression identifying the format of strings to apply this parser instance to. If a pattern is not specified, then parser is applied to all strings.
- **`StripQuotes`** – Flag indicating if double quotes (") surrounding a field value should be stripped. Should be set to `true` or `false`. Default is `true`.

This parser supports data in a variety of formats (with each format causing slightly different behavior of the parser. The support data formats are:

- Character string
- Character reader (`java.io.Reader`)
- Generic input stream (`java.io.InputStream`)

The parser internally requires a character string. When the data is supplied via a reader or stream, the object itself is not actually parsed, but instead the string is read from these (using a buffered line reader to read one line of data at a time).

 © 1998-2017 Nastel Technologies, Inc.

5.1.2 Name/Value Pair Parser

The Name/Value Pair parser splits the raw activity data into a list of tokens, with each separated by the given delimiter(s). Each token is assumed to be a name/value pair (e.g. name=value), and the locator is assumed to be the name.

The Name/Value Pair parser is implemented by the `com.nastel.janus.probe.directfeed.parsers.ActivityNameValueParser` class, and supports the following properties:

- **`FieldDelim`** – String of characters containing the field delimiters. Any character in this string is considered a delimiter. Default delimiter is `","` (comma).
- **`ValueDelim`** – String identifying the delimiter between name and its value. The string is considered a regular expression (based on `java.lang.String.split` method). Default delimiter is `"="` (equal).
- **`Pattern`** – Regular expression identifying the format of strings to apply this parser instance to. If a pattern is not specified, then parser is applied to all strings.
- **`StripQuotes`** – Flag indicating if double quotes (") surrounding a field value should be stripped. Should be set to `true` or `false`. Default is `true`.

This parser supports data in a variety of formats (with each format causing slightly different behavior of the parser. The support data formats are:

- Character string
- Character reader (`java.io.Reader`)
- Generic input stream (`java.io.InputStream`)

The parser internally requires a character string. When the data is supplied via a reader or stream, the object itself is not actually parsed, but instead the string is read from these (using a buffered line reader to read one line of data at a time).

5.1.3 Regular Expression Parser

The Regular Expression parser matches the raw activity data to the defined expression, extracting the group and/or match instance values.

The Regular Expression parser is implemented by the `com.nastel.janus.probe.directfeed.parsers.ActivityRegExParser` class, and supports the following properties:

- Pattern – Regular expression identifying the format of the raw activity data. This property is required.

This parser supports data in a variety of formats (with each format causing slightly different behavior of the parser. The support data formats are:

- Character string
- Character reader `(java.io.Reader)`
- Generic input stream `(java.io.InputStream)`

The parser internally requires a character string. When the data is supplied via a reader or stream, the object itself is not actually parsed, but instead the string is read from these (using a buffered line reader to read one line of data at a time).

5.1.4 XML Parser

The XML parser extracts XML element and attribute values from an XML document. The locators for each field are XPath expressions identifying a particular XML element or XML element attribute whose value is to be extracted.

The XML parser is implemented by the `com.nastel.janus.probe.directfeed.parsers.ActivityXmlParser` class, and supports the following properties:

- Namespace – defines a namespace prefix and its corresponding URI. This property is treated differently than the others, in that this property can be specified multiple times for different namespaces, with each defined namespace being added to a list of defined namespaces. The format of this field is: `prefix=uri`. An example definition is:

```
<property name="Namespace"
value="wmb=http://www.ibm.com/xmlns/prod/websphere/messagebroker/6.1.0/monitor
ing/event"/>
```

This parser supports data in a variety of formats (with each format causing slightly different behavior of the parser. The support data formats are:

- Character string
- Character reader (`java.io.Reader`)
- Generic input stream (`java.io.InputStream`)
- XML document object (`org.w3c.dom.Document`)

The parser internally requires an XML document object to do its parsing and to apply the XPath expressions. When the data is supplied as a character string, the string is assumed to be a complete XML document string, which is parsed into an XML document object. For reader and stream data, the object itself is not actually parsed, but instead the XML document string is read from these (using a buffered line reader) into a string buffer which is then parsed into an XML document. As a result, when feeding multiple XML document strings to the parser, each MUST be separated by a newline ("`\n`"). Each individual XML document may optionally contain newlines between its elements.

5.1.4.1 Messaging Activity XML Parser

The Messaging Activity XML Parser is an extension of the XML Parser specifically for supporting activity from a Messaging Server (e.g., WebSphere MQ, MSMQ, etc.). It allows for custom message signature calculation, which is used to uniquely identify a message. This is useful for linking activity observed by both Direct Feed probe and another TransactionWorks probe.

The XML parser is implemented by the `com.nastel.janus.probe.directfeed.parsers.MessageActivityXmlParser` class, and supports the same properties as the XML Parser.

5.2 Built-in Feeders

The Direct Feed Probe contains several built-in feeders for feeding data from some common data sources. Feeders can contain any number of parsers (at least one is required). The parsers are applied in the order specified until a parser successfully parses the raw activity data. If a required locator is not in the message being parsed, the parser skips it. The first parser to get all required locators will process the message. If no parsers accept the message, the message is skipped and the reader waits for the next message.

Each feeder returns the raw activity data in a specific format. Some feeders can return the data in multiple formats, although an instance of a feeder returns the data in one format. Feeders should only be used with parsers that support the data format they return the data in.

 © 1998-2017 Nastel Technologies, Inc.

5.2.1 Line-Oriented File Feeder

The Line-oriented File feeder is a simple feeder that reads one line from the specified file at a time and applies each of its parsers in turn to the current line read. This feeder is implemented by the `com.nastel.janus.probe.directfeed.feeders.FileLineFeeder` class, and supports the following properties:

- **`FileName`** – Name of the file to read raw activity data from.

This feeder returns the raw activity data as a character string, so is compatible with any parser that supports character string input, and therefore can be used with any of the built-in parsers.

5.2.2 Stream Feeder

The Stream feeder uses a stream or reader as the source of the raw activity data. The supplied stream or reader is wrapped with a buffered reader, thus converting byte streams to character streams. This feeder is implemented by the `com.nastel.janus.probe.directfeed.feeders.StreamFeeder` class, and supports the following properties:

- **`FileName`** – Name of the file to read raw activity data from, supplying a stream to the parser to read the raw activity data from.
- **`SocketPort`** – Port number to accept socket connections on, supplying the input stream for the socket to the parser to read the raw activity data from.

This feeder does not supply the actual raw activity data, but supplies a stream or reader from which the raw activity data can be read, so is compatible with any parser that supports streams or readers, and therefore can be used with any of the built-in parsers.

5.2.3 WMQ Feeder

The WMQ Feeder uses a WebSphere MQ queue as the source of the raw activity data. This feeder is implemented by the `com.nastel.janus.probe.directfeed.feeders.WmqFeeder` class, and supports the following properties:

- **`QueueManager`** – Name of the queue manager that the source queue is defined on.
- **`Queue`** – Name of the queue to read activity data messages from.
- **`Topic`** – Name of the topic object defining root of topic string.
- **`TopicString`** – Topic string to subscribe to.
- **`Subscription`** – Subscription object to use to subscribe to topic string.
- **`Host`** – For remote queue managers (queue managers not on same system as Direct Feed Probe), the name of the system where the queue manager is defined.
- **`Port`** – For remote queue managers, the port number for the listener (defaults to 1414).
- **`Channel`** – For remote queue managers, the name of the channel for making remote connections (defaults to SYSTEM.DEF.SVRCONN).
- **`StripHeaders`** – Flag indicating whether WMQ headers should be stripped from message before being passed to parser (defaults to: true).

This feeder returns the raw activity data as a character string, so is compatible with any parser that supports character string input, and therefore can be used with any of the built-in parsers.

Chapter 6: Applying Direct Feed Probe

Here we describe how the Direct Feed Probe can be applied in a variety of situations. The ***<tworks_home>*/probes/direct-feed/samples** directory contains the relevant files for the probe applications discussed here.

6.1 Log File Analysis

One application of the Direct Feed Probe is in the analysis of log entries. It can be used to "stitch" related log entries, both within a single log and, more importantly, across multiple logs. Additionally, the log entries can have various formats, both within the same log and across multiple logs.

The probe can be applied to logs that represent a series of activity representing a business transaction, like a web purchase or a stock trade. It can also be applied to system logs, to help correlate log entries spread across multiple files to aid in troubleshooting.

6.1.1 Simple Log File

As an example, let's consider sample log of purchase order activity (see ***<tworks_home>*/probes/direct-feed/samples/single-log** directory for complete example). Each entry (line) of the log represents some activity of the purchase order processing, and there is only a single log file. For format of each entry is a delimited string, with "|" as the delimiter, with the order of the fields being:

1. Activity date/time
2. IP Address where the order originated
3. Purchase Order number
4. Email address of user placing order
5. Purchase Order activity
6. Part ordered
7. Quantity ordered
8. Total Price

Some sample log entries are:

12 Jul 2011 12:34:52|111.222.123.210|10983|mrsmith@gmail.com|Order Placed|12-3456|12-3456|5|56.00

12 Jul 2011 12:44:23|111.222.123.210|10983|mrsmith@gmail.com|Order Received|12-3456|5|56.00

12 Jul 2011 13:22:36|221.112.133.233|11634|shopper54321@hotmail.com|Order Placed|12-73946|25|295.32

 © 1998-2017 Nastel Technologies, Inc.

The following is a possible Direct Feed Probe configuration for this log:

```
<?xml version="1.0" encoding="utf-8"?>
<tw-direct-feed
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="tw-direct-feed-probe.xsd">

  <parser name="TokenParser"
   class="com.nastel.janus.probe.directfeed.parsers.ActivityTokenParser">
    <property name="FieldDelim" value="|"/>
    <field name="StartTime" locator="1" format="dd MMM yyyy HH:mm:ss"/>
    <field name="ServerIp" locator="2"/>
    <field name="Correlator" locator="3"/>
    <field name="UserName" locator="4"/>
    <field name="ActivityName" locator="5"/>
    <field name="ActivityType" locator="5">
      <field-map source="Order Placed" target="START"/>
      <field-map source="Order Received" target="RECEIVE"/>
      <field-map source="Order Processing" target="OPEN"/>
      <field-map source="Order Processed" target="SEND"/>
      <field-map source="Order Shipped" target="END"/>
    </field>
    <field name="Value" locator="8"/>
  </parser>

  <feeder name="FileFeeder"
   class="com.nastel.janus.probe.directfeed.feeders.FileLineFeeder">
    <ta-conn protocol="TCP" host="localhost" port="6400"/>
    <property name="FileName" value="orders.log"/>
    <parser-ref name="TokenParser"/>
  </feeder>
</tw-direct-feed>
```

For this log, we have chosen the Tokenizer Parser, since the entries are simple token-delimited strings. We’ve set the field delimiter to “|”, and we’re using the defaults for the other properties. Since we have not defined a pattern, the parser is applied to every log entry.

The field entries map the items in the tokenized activity string to the appropriate activity field. The locator is the 1-based position of the field in the tokenized string. Here, we’re mapping the activity start time (and end-time since we’re not defining an **EndTime** field) to the first item in the string, and have indicated that the date/time entry has the specified format. Most of the other field definitions are just mapping additional fields to the appropriate fields.

Notice the **ActivityName** and **ActivityType** field definitions. Here we are not only using the purchase order activity description as the name of the activity, but we are also using it to map the activity onto one of the supported activity types based on the activity type.

However, the most important field setting here is the use of Correlator. In our example, we are setting the correlator to the purchase order number. This allows the TransactionWorks analyzer to group, or “stitch” together all the log entries corresponding to this purchase order as a single “transaction”.

We’ve also chosen the Line-oriented File feeder, since we’re processing a single file where each line of the file represents a distinct activity to report. The feeder has been configured to connect to the TransactionWorks Analyzer running on the current system, listening on port 6400, to read the raw activity data from the file named “orders.log”, and to use the parser instance named “TokenParser”.

To run the probe using this configuration, simply execute the following:

```
tworks-feed-probe -f:samples/single-log/single-log-cfg.xml
```

6.1.2 Multiple Log Files

As an example of using multiple log files, let's extend the sample purchase order activity above to have the messages stored in multiple log files (see ***<tworks_home>*/probes/direct-feed/samples/multiple-logs** directory for complete example).

Here, we use two files to store the purchase order information, one to store the activity for orders as they are received, and one to store the activity as they are processed. Since both files have the same format, we use the same parser to process them. The only change is that we define two feeders, one to process each file. The log entries will still be correlated using the order number.

6.2 Transaction Stitching

Another use case for the Direct Feed probe is when needing to join two apparently independent transactions into a single transaction. This can happen when a component in the Transaction Processing Network (TPN) does not have a probe installed, or due to the nature of the application being monitored, it's not possible for a probe to identify two messages as related.

In this example, we will use the Messaging Activity XML Parser (section 5.1.4.1, Messaging Activity XML Parser) to show how it can be applied to processing WebSphere MQ operations (see ***<tworks_home>*/probes/direct-feed/samples/tran-stitch** directory for complete example).

The data is reported to the Direct Feed probe as an XML string whose format is defined by `tw-msg-activity.xsd`, with a sample XML string defined in `tw-msg-activity.xml`. Here we map the activity properties with the application which is calling the WMQ function, as follows:

- **ServerName** – host name where application calling WMQ function is running
- **ApplName** – name of application calling WMQ function is running
- **UserName** – user under which application calling WMQ function is running under
- **OperationName** – WMQ function name
- **Correlator** – a UUID used to correlate this operation with another
- **EndTime** – time WMQ function completed. Here, the time is being reported as a numeric timestamp with millisecond resolution (e.g. result of `System.currentTimeMillis()`)
- **ElapsedTime** – time it took for the WMQ function to run, reported in microseconds
- **ResMgr** – Queue Manager name that operation was run at
- **ObjectName** – name of WMQ object (queue, channel, etc.) that operation was run on
- **ObjectType** – numeric value for MQOT_ defining object type
- **CompCode** – MQCC_ value returned by WMQ function
- **ReasonCode** – MQRC_ value returned by WMQ function
- Message-related fields:
    - **MsgData** – for MQGET and MQPUT, this the body of the message (or a portion of it)
    - The remaining fields come from the corresponding field in the Message Descriptor (MQMD)

 © 1998-2017 Nastel Technologies, Inc.

The Direct Feed probe configuration (tw-direct-feed-probe.xml) contains the parser definition for how the probe should interpret this message, mapping it onto the corresponding activity fields (and field values). We're not going to discuss every activity field here, since most of them are just simple one-to-one mappings. But we will discuss some of the others that are so simple:

- **`ActivityType`** – here, we determine the activity type based on the name of the operation using <field-map> definitions to map WMQ function name to appropriate TransactionWorks activity type.
- **`ResMgrType`** – here, we hard-code the resource manager type to "`MESSAGING_SERVER`", since resource manager is always a queue manager.
- **`ResType`** – here, we map WMQ object types, as defined by MQOT_ values, onto supported TransactionWorks resource type.
- **`StatusCode`** – here we mapped WMQ completion codes, as defined by MQCC_ values, onto supported TransactionWorks activity status codes.
- **`Signature`** – here is where the Messaging Activity XML Parser does its custom processing. The configuration builds a string concatenating each of the message field in the order specified with each item separated by the specified character sequence ("`#!#`"). Normally, this concatenated string would be the value used for this field. However, this parser instead extracts each item from this string and passes it to a function to compute a message signature (using method from probe API). It then uses this computed signature as the value for this field.

Chapter 7: Executing Direct Feed Probe

The Direct Feed probe can be executed by using the appropriate script:

- **`tworks-feed-probe.bat`**, for Windows
- **`tworks-feed-probe.sh`**, for Unix

Each script simply starts the probe, passing any command line arguments to the probe. The supported command line arguments are:

`-f:`*`<cfg_file>`*	Load Direct Feed probe configuration from the specified file. Default: `config/tw-direct-feed-probe.xml`
`-d`	Enable debug-level logging
`-t`	Enable trace-level logging.
`-h, -?`	Display usage message

The Direct Feed probe uses log4j for logging information during its execution, writing log messages to both the console and the file `log4j/tw-direct-feed-probe.log4j`. Note that trace-level messages are only written to the file. They are not written to the console.

 © 1998-2017 Nastel Technologies, Inc.

This page intentionally left blank.

Chapter 8: Building Custom Components

The Direct Feed probe can be extended by defining custom parsers and/or custom feeders. Here, we discuss the requirement that each must satisfy. See ***<tworks_home>*/probes/direct-feed/samples/custom** directory for sample custom components.

8.1 Building Custom Parsers

All parsers must extend the class `com.nastel.janus.probe.directfeed.parsers.ActivityParser`. For defining custom parsers, parser classes can either extend this class directly, or extend one of the built-in parsers. Here we'll discuss the required and optional methods that must be implemented when extending `ActivityParser` directly. Consult the Reference documentation reference for detailed descriptions of the classes and methods.

The following methods MUST be defined by all parsers:

- **parse** – this is the heart of the parser. It is the method that is invoked by feeders to convert the raw activity data into a TransactionWorks activity instance.
- **isDataClassSupported** – this method must indicate whether the parser can process activity data from the specified object (based on the class of object). This method is important, since feeders will first call this method to determine if the parser supports the activity data format it supplies the raw activity data in before invoking the `parse` method.

The following methods may be defined, overriding or extending the functionality supplied by the base `ActivityParser` class:

- **setProperties** – This method is called during processing of the configuration file, supplying all properties, along with their values, defined for this parser. All parsers should implement this method to process any custom properties, first calling method from base class for it to process any properties it supports.
- **addField** – This method is also called during processing of configuration file to add a field definition to the fields supported by this parser. In general, default implementation is sufficient.
- **getNextString** – This method will return the next character string from the specified raw activity object. The default implementation returns the supplied value directly, if it is already a string, or returns the next line from the supplied value if it is a character stream or reader.
- **applyFieldValue** – This method applies the given value to the specified field. This method can be overridden to provide custom processing of the extracted raw activity data for this field.

8.2 Building Custom Feeders

All feeders must extend the class `com.nastel.janus.probe.directfeed.feeders.ActivityFeeder`. For defining custom feeders, feeder classes can either extend this class directly, or extend one of the built-in feeders. Here we'll discuss the required and optional methods that must be implemented when extending `ActivityFeeder` directly. Consult the Reference documentation reference for detailed descriptions of the classes and methods.

Feeders implement the `java.lang.Runnable` interface, allowing each of them to be executed in a specific thread. The class `FeederThread` is designed to run feeders.

The following methods MUST be defined by all feeders:

- **getNextItem** – the core method for feeders. This method is responsible for extracting the next raw activity data instance from its input source to pass to each parser.

The following methods may be defined, overriding or extending the functionality supplied by the base `ActivityFeeder` class:

- **`setProperties`** – This method is called during processing of the configuration file, supplying all properties, along with their values, defined for this feeder. All feeders should implement this method to process any custom properties, first calling method from base class for it to process any properties it supports.
- **`getProperty`** – This method is used when a parser field is defined as the value for a particular feeder property. All feeders should implement this method to return the value of any custom properties, calling the method from the base class if it does not support or recognize the requested property.
- **`initialize`** – This method is called during thread startup to allow the feeder to perform any necessary initializations before it starts feeding raw data to the parsers. If this method is overridden, it must call the base class method to allow base class(es) to be properly initialized.
- **`addParser`** – This method is called during configuration processing to add a parser reference to this feeder. The default implementation is generally sufficient.
- **`getActivityPosition`** – This method is used during error processing to retrieve the position within the raw activity data source currently being processed (e.g. line number in file, etc.). Default implementation simply returns `0` (zero).
- **`getNextActivity`** – This method is responsible for retrieving the next raw activity item (via `getNextItem`), and invoking each parser in turn until one parser successfully parsers the raw data, returning the TransactionWorks activity item. The default implementation is generally sufficient.
- **`applyParsers`** – This method applies each parser in turn until one successfully parses raw activity data and returns a TransactionWorks activity item. The default implementation is generally sufficient.
- **`cleanup`** – This method is called when feeder thread is being stopped to allow feeder to release any resources it has. If this method is overridden, it must call the base class method to allow base class(es) to properly release any resources.
- **`run`** – the method representing the execution stream of the thread. The default implementation is generally sufficient.

8.3 Other Recommendations

When implementing custom parsers or feeders, the following are also recommended:

- Including logging using log4j – Each class should support error and debug logging using log4j, providing appropriate support for each logging level (ERROR, WARN, INFO, DEBUG, TRACE). The default log level used by Direct Feed Probe is INFO.
- Attempt to extend built-in parsers and feeders when possible. This allows custom classes to use the methods implemented by these classes, increasing the success of integrating the custom class.

8.4 Integrating Direct Feed Probe into Applications

While the Direct Feed probe is designed as a standalone application providing its own main method, it can also be integrated into new or existing applications. The steps necessary to integrate the Direct Feed probe into an application are:

- Create instance of `FeedConfiguration` class, which loads configuration and provides list of feeders and parsers.
- For each feeder in configuration, create a `FeederThread` to run it and start the thread.

For example:

```
FeedConfiguration cfg = new FeedConfiguration();
HashMap<String,ActivityFeeder> feeders = cfg.getFeeders();
for (Map.Entry<String,ActivityFeeder> f : feeders.entrySet()) {
   String feederName = f.getKey();
   ActivityFeeder feeder = f.getValue();
   FeederThread ft = new FeederThread(feeder, feederName);
   ft.start();
}
```

It is also possible to instantiate and configure the parsers and feeders without the configuration, but this advanced use is not discussed here. See reference documentation for detailed descriptions of classes and methods.

8.4.1 Integration with Log4j

If you use log4j as your logging framework, you can apply the parsing features of the Direct Feed probe to the log4j logging events and have them converted to activities and saved in the TransactionWorks Analyzer database.

Included with the Direct Feed probe is the log4j appender TnTParserAppender. It's an extension of the basic TnTAppender that comes with the TransactionWorks Probe SDK. See AutoPilot TransactionWorks Probe Developer's Guide for details on TnTAppender.

The TnTParserAppender is implemented by the class `com.nastel.janus.tnt4j.log4j.TnTParserAppender`. It applies the specified parser to the log4j event message text, extracting the activity fields from the log4j event, and sending the activity information to the analyzer. The parser must support character string input.

As an extension of the TnTAppender, the TnTParserAppender supports all the properties defined by TnTAppender. In addition, it also supports the following properties:

- `configFile` — Name of Direct Feed probe configuration file containing parser definition.
- `parserName` — Name of parser to use from configuration file
- `feederName` — Name of feeder to use from configuration file. This is only required if the parser is defined to reference properties in the feeder it is using.
- `ignoreParseErrors` — Flag indicating whether errors parsing the log4j event should be ignored (default: true). If the errors are ignored, then the event will still be sent to the analyzer, with the activity fields set based on the log4j event context. If errors are not ignored, then any parse errors will abort logging of the event to the analyzer. In either case, this has no effect on other appenders.

 © 1998-2017 Nastel Technologies, Inc.

A sample TnTParserAppender configuration is:

```
log4j.appender.tnt=com.nastel.janus.tnt4j.log4j.TnTParserAppender
log4j.appender.tnt.configFile=C:/nastel/AutoPilotM6/DirectFeed/config/tw-direct-
feed-probe.xml
log4j.appender.tnt.parserName=XmlParser
log4j.appender.tnt.ignoreParseErrors=false
log4j.appender.tnt.protocol=HTTP
log4j.appender.tnt.host=localhost
log4j.appender.tnt.port=6480
log4j.appender.tnt.accessToken=ABCDEFG
log4j.appender.tnt.locationInfo=true
log4j.appender.tnt.includeStackTrace=true
log4j.appender.tnt.Threshold=TRACE
```

Appendix A: References

A.1 Nastel Documentation

The following table provides a list of reference information required for using the AutoPilot TransactionWorks Probes.

Table A-1. Nastel Documentation

Document Number (or higher)	Title
AP/JBOSS 410.001	*AutoPilot Plug-in for JBoss Guide*
M6/DSB 610.003	*AutoPilot M6 Business Dashboard*
M6/INS 600.008	*AutoPilot M6 Installation Guide*
M6/USR 600.011	*AutoPilot M6 Administrator's Guide*
M6/WS/JMX 600.002	*AutoPilot Plug-in for WebSphere Application Server (JMX) Guide*
TW/E ADM 650.002	*AutoPilot TransactionWorks Explorer Administrator's Guide*
TW/E USR 650.002	*AutoPilot TransactionWorks Explorer User's Guide*
TWORKS/NP 650.001	*AutoPilot TransactionWorks .NET Probe Installation and User's Guide*
TWORKS/P 650.001	*AutoPilot TransactionWorks Java Probes Installation and User's Guide*
TWORKS/TA 650.001	*AutoPilot TransactionWorks Transaction Analyzer Installation and User's Guide*
TWORKS/TP 650.001	*AutoPilot TransactionWorks Probe for WebSphere MQ (Distributed) Installation, Configuration and User's Guide*
TWORKS/WSz 650.001	*AutoPilot TransactionWorks Probes for z/OS Installation and User Guide*

A.2 Documentation

http://logging.apache.org/log4j/

 © 1998-2017 Nastel Technologies, Inc.

This page intentionally left blank.

Appendix B: Conventions

B.1 Typographical Conventions

Table B-1. Typographical Conventions

Convention	Description
Blue/Underlined	Used to identify links to referenced material or websites. Example: support@nastel.com
Bold Print	Used to identify topical headings, glossary entries, and toggles or buttons used in procedural steps. Example: Click **EXIT**.
Italic Print	Used to place emphasis on a title, menu, screen name, or other category.
Monospaced bold	Used to identify keystrokes/data entries, file names, directory names, etc.
Monospaced italic	Used to identify variables in an address location. Example: [*C:\AutoPilot_Home*]\documents, where the portion of the address in brackets [ ] is variable.
Monospaced text	Used to identify addresses, commands, scripts, etc.
Normal Text	Typically used for general text throughout the document.
Table Text	Table text is generally a smaller size to conserve space.

 © 1998-2017 Nastel Technologies, Inc.

This page intentionally left blank.

Glossary

Application: A logical collection of software components that perform a business function**,** running on a specific server.

AutoPilot M6: Nastel Technologies' Enterprise Application Management Platform. AutoPilot monitors and automates the management of *e*Business integration components such as middleware application, application servers and user applications.

AutoPilot/Message Tracking (AP/MT): Nastel's AutoPilot/Message Tracking plug-in that enables AutoPilot to intercept message exits and forward the statistical data to an AutoPilot expert.

AutoPilot TransactionWorks (AP/TW): Nastel Technologies' transaction and application performance monitoring product.

AutoPilot/WebSphere Message Queue Integrator (AP/WMQI): Formerly AP/MQSI.

BCI: *See* Byte Code Instrumentation.

BSV: *See* Business Views.

Business Transaction: A collection of related Transactions that comprise a user defined business function (for example, purchase a book, return merchandize, purchase stock). Each of the business activities may be comprised of various workloads.

Business View (BSV): A collection of rules that define a desired state of an eBusiness environment. Business Views can be tailored to present information in the form most suited to a given user, as defined by the user.

Byte Code Instrumentation (BCI): The process of adding small portions of Java byte code around methods of a Java class. The added code performs tasks such as time spent or CPU utilization within the monitored class.

CEP: *See* Complex Event Processing.

CEP Server: A container that can host any number of AutoPilot services such as experts, managers, policies, etc. (called managed node prior to AutoPilot M6 Service Update 6.)

Client: Any programming component that uses the AutoPilot infrastructure; for example, the AutoPilot Console.

Common Object Request Broker Architecture (CORBA): A standard defined by the Object Management Group that enables software components written in multiple computer languages and running on multiple computers to work together. It can be invoked from a Web browser using CGI scripts or applets.

Complex Event Processing (CEP): Primarily an event processing concept that deals with the task of processing multiple events from an event cloud with the goal of identifying the meaningful events within the event cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing, and event-driven processes.

Composite Application: A collection of applications that collaborate or communicate with each other (have related sessions).

Console: The console acts as the graphical interface for AutoPilot.

Contacts: A subordinate to a given Manager or Expert.

CORBA: *See* Common Object Request Broker Architecture.

 © 1998-2017 Nastel Technologies, Inc.

Data Source Name: A Data Source Name (DSN) is the logical name that is used by Open Database Connectivity (ODBC) to refer to the drive and other information that is required to access data. The name is use by Internet Information Services (IIS) for a connection to an ODBC data source, (example: Microsoft SQL Server database). The ODBC tool in Control Panel is used to set the DSN. When ODBC DSN entries are used to store the connection string values externally, you simplify the information that is needed in the connection string. This makes changes to the data source completely transparent to the code itself.

Deploy: To put to use, to position for use or action.

Derby Database Server: A relational database management system that is based on Java and SQL. It will run in any certified Java Virtual Machine.

Domain Server: A specialized managed node that maintains the directory of managed nodes, experts etc. The domain server is also capable of hosting experts, managers, etc.

DSN: *See* Data Source Name.

Event: An *Event* is something that happens to an object. Events are logged by AutoPilot and are available for use by AutoPilot Policies or the user.

EVT: Event Log file extension (for example, `sample.evt`).

Expert: Services that monitor specific applications such as an applications server, Web server or specific components within the applications (for example, channels in WMQ). Experts generate facts.

Fact: Facts are single pieces of data that has a unique name and value. One or more facts are used to determine the health of the object, application or server.

Graphical User Interface (GUI): A type of environment that represents programs, files, and options by means of icons, menus, and dialog boxes on the screen. The user can select and activate these options by pointing and clicking with a mouse or, often, with the keyboard. Because the graphical user interface provides standard software routines to handle these elements and report the user's actions (such as a mouse click on a particular icon or at a particular location in text, or a key press); applications call these routines with specific parameters rather than attempting to reproduce them from scratch.

GUI: *See* Graphical User Interface.

Heap: In Java programming, a block of memory that the Java virtual machine uses at run time to store Java objects. Java heap memory is managed by a garbage collector, which automatically de-allocates Java objects that are no longer in use.

IIS: *See* Internet Information Services.

Instrumentation: Modifies a program by adding code at particular program points to capture dynamic information. For example, a program could be instrumented to count how many times each method is called.

Internet Information Services: Microsoft's brand of Web server software, utilizing HTTP to deliver World Wide Web documents. It incorporates various functions for security, allows CGI programs, and also provides for Gopher and FTP services.

Java: A platform-independent, object-oriented programming language developed and made available by Sun Microsystems.

Java Database Connectivity (JDBC): The JDBC API provides universal data access from the Java programming language. Using the JDBC 2.0 API, you can access virtually any data source, from relational databases to spreadsheets and flat files. JDBC technology also provides a common base on which tools and alternate interfaces can be built. The JDBC Test Tool that was developed by Merant and Sun Microsystems may be used to test drivers, to demonstrate executing queries and getting results, and to teach programmers about the JDBC API.

Java Developer's Kit (JDK): A set of software tools developed by Sun Microsystems, Inc., for writing Java applets or applications. The kit, which is distributed free, includes a Java compiler, interpreter, debugger, viewer for applets, and documentation.

Java Messaging Service (JMS): A Java Message Oriented Middleware API for sending messages between two or more clients.

Java Platform, Enterprise Edition (Java EE): The industry standard for developing portable, robust, scalable and secure server-side Java applications. Building on the solid foundation of Java SE, Java EE provides Web services, component model, management, and communications APIs that make it the industry standard for implementing enterprise class service-oriented architecture (SOA) and Web 2.0 applications.

Java Server Pages (JSP): JSP technology enables rapid development of Web-based applications that are platform independent. Java Server Pages technology separates the user interface from content generation enabling designers to change the overall page layout without altering the underlying dynamic content. Java Server Pages technology is an extension of the JavaTM Servlet technology.

Java Transactions API (JTA): Specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications.

Java Virtual Machine (JVM): The "virtual" operating system that JAVA-written programs run. The JVM is a hardware- and operating system-independent abstract computing machine and execution environment. Java programs execute in the JVM where they are protected from malicious programs and have a small compiled footprint.

JDBC: *See* Java Database Connectivity.

JDK: *See* Java Developer's Kit.

JMS: *See* Java Messaging Service.

JRE: Java Run-time Environment. The minimum core Java required to run Java programs.

JSP: *See* Java Server Pages.

JTA: *See* Java Transactions API.

JVM: *See* Java Virtual Machine.

Logical Unit of Work (LUW): A collection of operations and messages within a session that should be considered to be a single unit of work (all or nothing property). These are generally delimited by BEGIN/COMMIT calls.

LUW: *See* Logical Unit of Work.

M6 for WMQ: Nastel Technologies' WebSphere MQ management solution. Re-designated as M6 for WMQ (formally known as AutoPilot M6 for WMQ) with release 6.0. Prior releases retain either AP/WMQ for version 4.0 or MQControl trademark for version 3.0 and prior.

M6 Web Server: M6 Web Server is a browser-based interface that provides monitoring and operational control over managed resources and applications.

Managed Node: A container that can host any number of AutoPilot services such as experts, managers, policies, etc. (Managed node changed to CEP Server with AutoPilot M6 Service Update 6.)

Manager: Managers are the home or container for policies. All business views must reside on managers, and manager must be deployed prior to deploying a business view or policy.

Message: A physical message being transported through the TPN.

Message-Oriented Middleware (MOM): A category of inter-application communication software that relies on asynchronous message passing as opposed to a request/response metaphor.

Message Queue Interface: The Message Queue Interface (MQI) is part of IBM's Networking Blueprint. It is a method of program-to-program communication suitable for connecting independent and potentially non-concurrent distributed applications.

MOM: *See* Message-Oriented Middleware.

MQControl: Nastel Technologies' MQSeries management product. Re-designated as AutoPilot/MQ with release 4.0, prior releases retain the MQControl trademark.

MQI: *See* Message Queue Interface.

MQSeries: IBM's message queuing product; renamed by IBM as WebSphere MQ.

© 1998-2017 Nastel Technologies, Inc.

Naming Service: A common server records "names" of objects and associates them with references, locations and properties.

ORB: Object Request Broker.

Orbix: CORBA product distributed by IONA Technologies.

Package Manager: The command line utility that allows users to list, install, uninstall, verify and update AutoPilot installation on any Managed Node.

PKGMAN: *See* Package Manager.

Policy/Business Views: Business views are a collection of one or more sensors. Business views are used to visually present the health and status of the different systems as well as automatically issue remedial actions.

Resource: An entity on which transactions are executed or a medium of exchange. Examples include Queue, DB table, file, JMS topic.

Resource Manager: An entity that is managing a collection of resources. Examples include a WMQ Queue Manager, Application Server, Database Server.

Sensor: A rule that is used to determine the health of an object or application based on one or more facts. Actions can then be issued, based on the health.

Server: A physical or virtual node within **a** TPN that hosts all transaction processing activity**.**

Session: A specific period of execution of an application. Examples include the interval during which a database or queue manager connection is active.

Simple Mail Transfer Protocol (SMTP): A TCP/IP protocol for sending messages from one computer to another on a network. This protocol is used on the Internet to route e-mail. See also communications protocol, TCP/IP. Compare CCITT X series, Post Office Protocol.

SMTP: *See* Simple Mail Transfer Protocol.

Speed Manager: Type of manager which allows loading of policies from a "Speed Folder" which automatically loads all `.bsv` and `.bsp` files located in the folder upon manager's start.

TCP/IP: *See* Transmission Control Protocol/Internet Protocol.

Transaction: A group of activities targeted at achieving a common goal or a task. Collection of related sessions and LUWs.

Transmission Control Protocol/Internet Protocol (TCP/IP): A protocol developed by the Department of Defense for communications between computers. It is built into the UNIX system and has become the de facto standard for data transmission over networks, including the Internet.

Virtual Machine: Software that mimics the performance of a hardware device, such as a program that allows applications written for an Intel processor to be run on a Motorola chip. *Also see* Java Virtual Machine.

WebLogic: A Java EE compatible application server platform which enables support for multiple programming models, which includes advanced administration tools and is the ideal foundation for Service Oriented Architecture (SOA).

WebSphere MQ: IBM's message queuing product; formerly known as MQSeries.

Websphere_MQ_Manager: A specialized manager capable of hosting one or more WebSphere MQ specific policies, apart from the regular policies.

Wireless Application Protocol (WAP): An open global specification that is used by most mobile telephone manufacturers. WAP determines how wireless devices utilize Internet content and other services.

Index

B

built-in feeders 19
built-in parsers 17

C

configuring feeders 15
configuring parsers 10

D

definitions 6
Direct Feed Probe 4
distribution 7
document history 1

E

environment variables 9

F

feedback, user 2
feeders
- building custom 27
- built-in 19

I

installation 7
- requirements 7

M

monitoring 4

O

operating standards 7

P

parsers
- building custom 27
- built-in 17
- tokenizer 17

probes 3

R

RAM requirements 7
README files 2
requirements, minimum 7
Resource Center 2

S

sample applications 27
stitching 23
support@nastel.com. 2
system requirements 2

T

technical support 2
Transaction Analyzer
- conponents 3
- features 3
- transport options 3

TransactionWorks
- architecture 3
- Explorer 3
- Transaction Analyzer 3
- Transaction Probes 3

 © 1998-2017 Nastel Technologies, Inc.

This page intentionally left blank.