Technical white paper

Introduction to the HP NonStop Server for Java ecosystem

Table of contents

Click here to verify the latest version of this document

Abstract

This document summarizes the components of the HP NonStop H- and J-Series Servers for Java ecosystem and how they integrate with core NonStop software products such as NonStop SQL, NonStop Transaction Services (TS/MP), and the Transaction Monitoring Facility (TMF). It also provides some performance influencers and how to use the NonStop performance-measuring tool MEASURE, with Java applications. This document references documentation related to the HP NonStop Server for Java (NSJ) ecosystem. This document is intended for architects, designers, and developers of Java applications being written for or ported to the HP NonStop platform. Those who install or manage NonStop Java applications may also find this document useful. This document assumes familiarity with NonStop servers and/or Java.

Note
Unless otherwise noted, documentation identified in the "References" subsections in this paper is available from the NonStop Technical Library at hp.com.[1]

Introduction

The HP NonStop platform offers an open application development environment and Java Virtual Machine and tools. Java is used on the NonStop platform to:

- Develop and deploy Java applications and support third party Java applications.
- Host enterprise-class application servers and containers, messaging infrastructure, development tools, and frameworks.
- Access the NonStop SQL/MX database.
- Implement Stored Procedures in Java (SPJs)

The NonStop Server for Java implementation provides hooks to integrate Java applications with the platform's parallel, shared-nothing architecture.

HP NonStop Server for Java Standard Edition 7.0 software

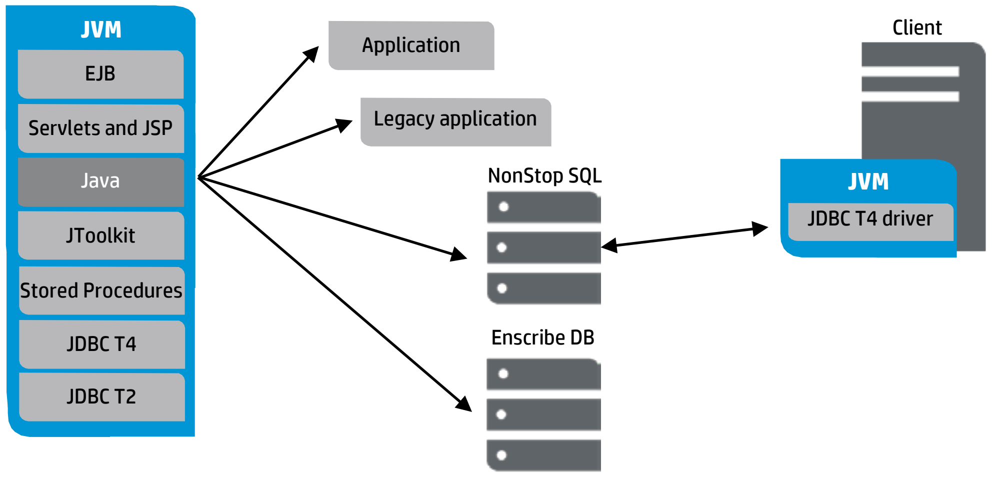
HP NonStop Server for Java Standard Edition 7.0 (NSJ7) provides a Java environment that supports compact, concurrent, and dynamic portable programs running on HP NonStop Server systems in the NonStop Open System Services (OSS) environment. NSJ7 is a fully compliant headless Java Development Kit (JDK) based on the Java Platform Standard Edition (Java SE) 7.0 reference and offers the standard JDK toolkit. The NonStop Java JToolkit allows Java applications to leverage NonStop availability, scalability, and data integrity fundamentals.

NSJ7 is available in 32–bit (T2766) and 64–bit (T2866) product components. On 64-bit Java Virtual Machines (JVMs) running on multi-core enabled systems, parallel and CMS garbage collection algorithms can be used to reduce the application pause time. NSJ7 implements the HotSpot server compiler and the runtime Java HotSpot virtual machine. Java Naming and Directory Interface (JNDI) functionality is available. NSJ7 supports non-blocking I/O for OSS files. JDBC drivers provide connectivity to the NonStop SQL/MX database and other SQL databases. The SQL/MX database engine implements Stored Procedures in Java (SPJ). These user-defined routines are written as Java methods contained in Java classes and registered in SQL/MX metadata tables.

Note
SQL/MX is a clustered, shared-nothing, scalable SQL implementation designed specifically for the NonStop platform. This document includes a discussion on which driver to choose and describes how the database is accessed using JDBC Type 2 or JDBC Type 4 via the SQL/MX Connectivity Service MXCS.

[1] To access the NonStop Technical Library, browse to hp.com/go/nonstop-docs and select the appropriate server model (H-Series or J-Series). Titles and release numbers referenced in this paper are as they appear in the J-series Technical Library at the time of this paper's publication.

Figure 1. Java-based products on NonStop systems.



Architectural qualities of the NonStop Server

While the NonStop Server is an open platform that conforms to standards, it offers extraordinary capabilities in scalability, availability and performance. Most Java applications are executed on Symmetrical Multi-processor (SMP) servers that consist of multiple processors—all sharing the same memory space for code and data. Java applications run within a JVM that can use all available processors and memory to achieve scalability.

A NonStop Server is a message-passing cluster of independent and relatively small servers, each having their own processors and memory not shared with other servers. This architecture is referred to as a Massively Parallel Platform (MPP) that can be combined into a “super cluster” containing up to 4080 processors. Communication between processes occurs via a message system that virtualizes all resources into a single system image. Individual server instances are small, so the same workload that is executed within one JVM on an SMP platform may be handled by multiple JVMs running on a NonStop server.

The NonStop Server’s architectural qualities of scalability, availability and load balancing are not dependent on the language that is used to develop applications. Most NonStop middleware and traditional applications are implemented as TS/MP server classes. HP Pathway with NonStop TS/MP software provides a built-in application environment replete with load-balancing, communications I/O, memory management, fault tolerance, and threading and scheduling. A TS/MP server class is a named, logical entity representing a pool of logically identical server processes configured and managed as a group.

It is possible to code Java servers using TS/MP APIs and native NonStop inter-process communication (IPC) and run them directly in a Pathway environment. However, the more common (and standard) approach is to deploy and execute NSJ applications in JEE application containers under the management of TS/MP. Thus, the scaling of Java applications on the NonStop platform involves adding multiple JVMs, running the same applications and forming a cluster or pool of JVMs that are managed as a single logical entity in TS/MP. TS/MP management also enhances availability and facilitates load-balancing and other operations management tasks.

References
For more information on HP Pathway with NonStop TS/MP software go to: hp.com/V2/GetDocument.aspx?docname=4AA0-3689ENW&cc=us&lc=en.

Alternatively it can be accessed by navigating to hp.com/go/nonstop → click on Products tab → scroll down and click on Middleware and Java hyperlink → and scroll down and click on Data sheets section.

Scalability

Scalability is achieved by scale-out of an application across multiple instances of processes, executing in multiple processors or even multiple nodes. The NonStop Server has an atypical architecture that supports high levels of availability and scalability and, in the case of Java, also affects how scaling occurs. A JVM runs in one logical CPU of the server. The JVM can process multiple threads at a time, but all application threads run in the same core of the processor, even if there are multiple cores in that processor. Currently, the other cores are used by the JVM instance for Garbage Collection threads and by the operating system (OS). Therefore, the limit to the amount of work a single JVM can perform is lower than what typically occurs with other host OSs. However, it does not mean the other JVMs are idling and hence are wasted. They reduce the load on the application running JVM by taking care of the garbage collection functionality.

Scalable Java on NonStop is achieved by configuring a pool of JVMs (running the same application) in Application Containers using the Transaction Services/Massively Parallel (TS/MP) middleware.

Availability

Application availability is derived from the same architectural principles that provide scalability. There are multiple application processes, all capable of providing the same application services. Therefore, in the event of a failure that brings one or more of these instances down (for example, when a process or processor failure occurs), there are many processes left that can service the requests. The message system and TS/MP will automatically route incoming requests to the remaining instances and if necessary, new instances will be started when needed.

Load balancing

Balancing the load across all processors of the NonStop server is a key feature in achieving scalability and availability. The process instances are started in all processors by the OS either automatically or manually by the user or operator through appropriately configured applications. The incoming requests are distributed among these processes based on the type of request. For example:

- TCP/IP requests arriving through ServerNet and the TCP/IP v6 stack are distributed across multiple listener processes running in multiple CPUs. These processes can share ports so all processes can deliver the same service. Adding processing capacity, i.e. increasing the number of listening process instances, will provide more throughput (the scalability aspect) and a processor failure will cause requests to be sent to other processes listening on the same port (the availability aspect).
- Database requests that come from off-platform clients using the ODBC or JDBC Type 4 drivers are served by the NonStop SQL/MX Connectivity Service, which distributes the requests across configured SQL/MX server instances in multiple processors.
- Requests coming from applications that execute in application containers hosted on the NonStop server are managed through the TS/MP router mechanism.

A special kind of load balancing is done where data is concerned. Data stored on disk is organized within logical volumes. A volume can be:

- A physical hard disk drive (HDD) or a solid state drive (SSD)
- A part of a partitioned disk or a logical unit (LUN) on an Enterprise Storage device
- A Storage Area Network (SAN)

Volumes are managed by a Disk Access Manager (DAM) and usually mirrored for fault tolerance. This means that the DAM keeps two copies of the data; a primary copy and a mirror copy. When a volume becomes a hotspot due to too many requests being serviced by the device, the data can be partitioned across multiple volumes transparently to the application. The NonStop file system automatically routes requests for data to the appropriate DAMs.

Application Containers

Application containers provide an environment in which a business application executes. The environment provides business applications with services such as ensuring continuous availability of the application, security, data services lifecycle (some containers include transaction management as well), and performance instrumentation. Similar terms for this technology on other platforms are Transaction Monitors and Application Servers. Traditionally, these services have been supplied on NonStop Servers by the products TS/MP and NonStop TMF, where TS/MP provides both process management and messaging API and NonStop TMF ensures cluster-wide transaction integrity.

Currently, TS/MP allows development of services in the Java language using the same (proprietary) APIs that the traditional applications written in COBOL or C/C++ also use. In addition to this, the NSJSP (Apache® Tomcat® web container), the NSASJ (JBoss Application Server) as well as the iTP Webserver and the NonStop Simple Object Access Protocol (SOAP) products make use of the TS/MP infrastructure.

TS/MP (Pathway) – Private Container

Many applications developed in COBOL, C/C++, or the NonStop-specific pTAL language for NonStop systems use the services of TS/MP. TS/MP allows applications to be written as services, which can be deployed across multiple processors in the system, and even across NonStop nodes in a NonStop cluster. These services are managed by the TS/MP software that routes requests to free instances of a server class (a logical grouping of services), and starts and terminates instances of these server classes as needed based on a specified configuration. The services are invoked by a client via a specific API call that uses a logical server class name. The object that processes the request simply reads the request from a system-defined queue, called $RECEIVE, also using a specific API call.

The business code of the service is then executed in a single-threaded manner as follows:

- Read a request from the queue
- Process it by executing database calls
- Possibly execute other service calls
- Reply with the required data to the requestor

Neither the client nor server side needs to be concerned with starting processes, locating services, load balancing or handling exceptions that cause processes to fail. Clients, however, are required to send data in a format that the server understands. This functionality does not exist in the standard Java APIs. However, the NonStop Server for Java provides a toolkit (JToolkit) that includes classes to invoke the send and receive API calls, the Pathsend API, and classes that can be used to convert Java data objects to and from TS/MP messages. This toolkit allows Java programs to interface with existing TS/MP applications as clients, but also to be implemented as TS/MP services.

References
The TS/MP Pathsend and Server programming manual provides an overview of the traditional NonStop requester-server programming model including the API calls. However, it does not contain references to clients or servers written in Java. For that information refer the documents listed below:

- *HP NonStop Server for Java 7.0 Programmer's Reference, Chapter 1: Introduction to NSJ7*
- *JToolkit reference pages can be downloaded from the HP Software Depot.*
  hp.com/portal/swdepot/displayProductInfo.do?productNumber=NS-API
- *TS/MP 2.5 Pathsend and Server Programming Manual*, Chapter 1: Introduction to Pathway Application Programming Java

NSJSP (Tomcat implementation) – Web Container

NonStop Servlets for Java Server Pages (NSJSP) provides Web Container functionality on NonStop servers. NonStop Servlets is an implementation of the Apache Tomcat, NSJSP is to be installed in a NonStop iTP Secure Webserver environment where it uses the web server to handle the HTTP(S) protocol. The container provides an environment in which one can deploy, execute and manage web applications based on servlets or Java Server Pages. The communication between iTP Webserver and the NSJSP processes uses a dedicated NonStop connector for NSJSP to allow native Inter Process Communication (IPC). The native IPC allows the NSJSP processes to be defined as TS/MP server classes that share a common configuration. This enables load balancing of requests across all the processes in the server class and allows placement of the NSJSP processes in multiple NonStop processors to support scalability and application availability. The number of processes can be increased statically or dynamically, allowing significant scaling of the web application capacity. At the time of this document's publication, NSJSP was based on Apache Tomcat version 7.0.10. Readers are advised to refer to the latest version of *NonStop Servlets for JavaServer Pages (NSJSP) 7.0 System Administrator's Guide* to know the current Tomcat version on which NSJSP is based on.

References
- *NonStop Servlets for JavaServer Pages (NSJSP) 7.0 System Administrator's Guide,* Chapter 1: Introduction to NSJSP
- *iTP Secure WebServer System Administrator's Guide*, Chapter 1: Introduction to the iTP Secure Webserver

NSASJ (JBoss implementation) – EJB and Web Container

NSASJ is a Java EE Application Server that provides an implementation of a subset of Java EE technologies. It includes the support for certain JEE specified functionalities. At the time of this document's publication, NSASJ was based on JBoss AS version 7.1.2. Refer to the latest version of NonStop Application Server for Java User Guide to know the current version of JBoss AS that NSASJ is based on and to know the JBoss Functionalities and modules supported in NSASJ. NSASJ provides an EJB container hosting Enterprise Java Beans that applications (running stand-alone or in web containers) can invoke. For web applications running on NonStop, the web container can be NSJSP (described in the previous section), or customers may choose to deploy web applications in NSASJ. NSASJ is a port of the popular open source JBoss Application Server. NSASJ is well integrated with platform technologies such as TS/MP and TMF, and NonStop SQL. Both remote and on-platform clients can use the services of the applications running in the NSASJ container. It provides secure JBoss Remoting and HTTP(S) interfaces to remote clients.

References
- *NonStop Application Server for Java User Guide*, Chapter 1: Introduction; Chapter 2: Features of NSASJ

Java application frameworks

Open Source frameworks can provide an environment that enables the development of Enterprise Java applications using standard, open technologies. At the time of this document's publication, we have certified the following frameworks for use on NonStop systems:

- Spring 3.1.0
- Hibernate 4.1.1
- Apache MyFaces 2.0.2
- Apache Axis2/Java 1.5.2

Refer to the latest version of *Open Source Java Frameworks on NonStop User's Guide* to know the current versions certified on NonStop.

When used to build web applications, these frameworks typically run in the NSJSP or NSASJ containers, which provide the NonStop fundamentals of scalability and availability. Hibernate and Spring can also be used to develop non-web Java applications. The following sections describe frameworks certified with NonStop systems.

Spring framework

Spring provides a lightweight, open source framework for implementation of business logic in enterprise applications. Spring has a layered architecture, and can also serve as an abstraction layer integrated with other frameworks such as Hibernate. Spring also supports lightweight remote support.

Spring's features and functions are packaged in separate modules that can be combined in a lightweight container that provides centralized, automated configuration and wiring of application objects. It includes a flexible MVC web application framework that is configurable via interfaces and accommodates multiple view technologies, like JSF. Spring also facilitates enterprise application development by allowing software components to be first developed and tested in isolation and then scaled up for deployment.

References
- *Open Source Java Frameworks on NonStop User's Guide*, Chapter 2: Spring Framework

Hibernate framework

Hibernate is an object-relational mapping (ORM) tool for incorporating database access in Java applications. It provides a framework for mapping data representation between an object model (Java class) and a relational (SQL) data model. Hibernate scales well in any environment and is highly extensible and customizable. In addition to configuration files for data mapping, Hibernate supports property-based definitions of database URLs, database credentials, connection pooling and other configuration parameters. It also supports lazy initializations, fetching strategies, and optimistic locking with automatic versioning and time-stamping. HP provides and supports a Hibernate dialect for NonStop SQL/MX.

References
- *Open Source Java Frameworks on NonStop User's Guide, Chapter 3:* Hibernate Framework

Apache MyFaces framework

Apache MyFaces is an event-based framework used to design user interfaces to web applications. Its components support validation of user inputs and the use of Tiles and Converters. The core of the NonStop version implements the JavaServer Faces (JSF) specification. Subproject components add features that work with the core or any other implementation of JSF specifications.

References
- *Open Source Java Frameworks on NonStop User's Guide,* Chapter 4: MyFaces Framework

Apache Axis2/Java framework

Apache Axis2/Java is a high speed, flexible framework for developing web services using the SOAP protocol and WSDL (Web Services Description Language) files. The Axis2/Java framework enables applications to send, receive, and process SOAP messages with or without attachments, create a Web service from a plain Java class, create implementation classes for both server and client, and create and utilize REST based web services. This framework is scalable and includes features such as hot deployment. Axis2/Java can plug into servlet engines as a server and includes tools that can be used to create Java client and servers from the WSDL file.

References
- *Open Source Java Frameworks on NonStop User's Guide, Chapter 5:* Axis2/Java Framework

Java access to the NonStop SQL database

The following two functionally equivalent Java Database Connectivity (JDBC) drivers provide API access to NonStop SQL/MX databases:

- Type 2 driver (T2) – uses native SQL/MX calls and can be invoked only from the NonStop Server for Java Virtual Machine (the JVM).
- Type 4 driver (T4) – written in Java, can be invoked from any platform that supports a JVM – including the NonStop server.

The discussion in this section is limited to first describing and then comparing the drivers. Tuning is discussed separately in the JDBC driver tuning considerations section.

Support for large objects

Although the maximum row size is 32 KB, SQL/MX supports Character Large Objects (CLOBs) and Binary Large Objects (BLOBs) via the JDBC drivers. These drivers support the JDBC API calls but manage the objects in tables dedicated to large objects.

References
- *JDBC Type 2 Driver Programmer's Reference for SQL/MX,* Chapters 4 and 5: Working with BLOB and CLOB Data, Managing the SQL/MX Tables for BLOB and CLOB Data
- *JDBC Type 4 Driver Programmer's Reference for SQL/MX,* Chapters 5 and 6: Working with BLOB and CLOB Data, Managing the SQL/MX Tables for BLOB and CLOB Data

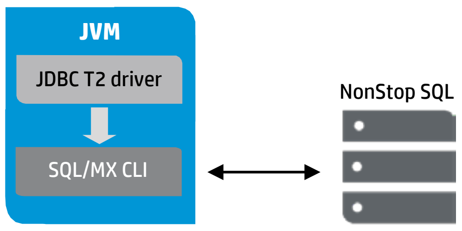
JDBC Type 2 driver access

The JDBC Type 2 (T2) driver uses native SQL/MX calls and can be invoked only from the NonStop Server for Java JVM. When the T2 driver is invoked by a JVM running on the NonStop server, it employs native APIs (also known as the SQL/MX CLI – Call Level Interface) to access the NonStop SQL database. Because no other subsystems are involved, this provides the shortest path to the data – and the highest form of availability.

Decision points for use of the T2 driver include:

- The T2 driver can be used only by Java programs running on the NonStop server.
- The T2 driver *must* be used to access SQL/MX from within Stored Procedures (SPJs), which are written in Java. SPJs are discussed in the section Stored Procedures in Java.

Figure 2. JDBC T2 driver architecture.



The T2 driver conforms, where applicable, to the Oracle JDBC 3.0 API specifications. Exceptions are noted in the Programmer's Reference manual.

References
- JDBC Type 2 Driver Programmer's Reference for SQL/MX, Chapter 1: Introduction to JDBC/MX Driver
- JDBC Type 2 Driver Programmer's Reference for SQL/MX, Chapter 7: JDBC/MX Compliance

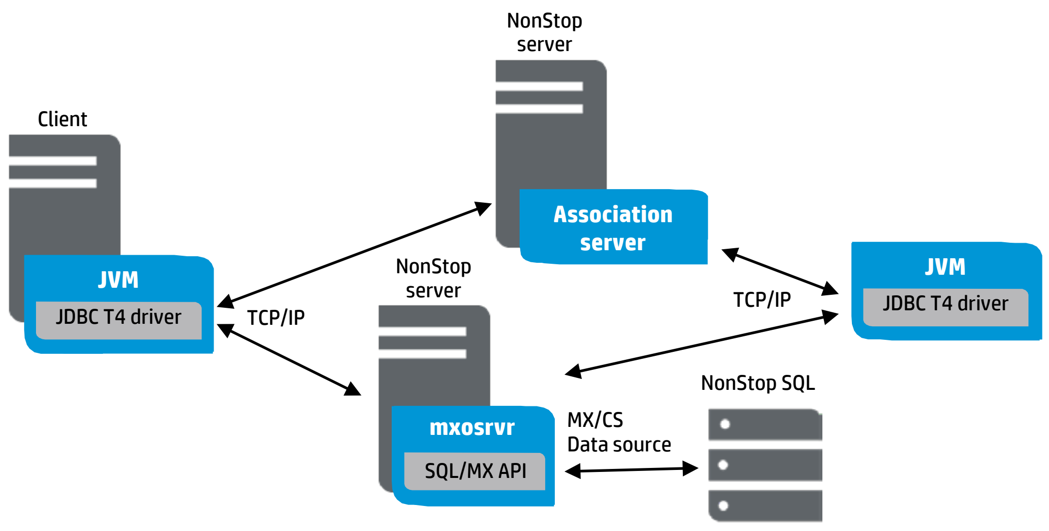
JDBC Type 4 driver access

The Type 4 (T4) driver, written in Java, is a "pure Java" driver that can be invoked from any platform that supports a JVM, including the NonStop server (locally and remotely) and platforms running Linux, UNIX or Windows. The T4 driver also differs from the T2 driver in how it connects to the database. Unlike the tightly integrated T2 driver, T4 driver connections go through a separate software subsystem, the SQL/MX Connectivity Service (MXCS). MXCS authenticates each client request against the NonStop operating system's Safeguard user database before granting a database connection. Connections are granted to a data source, which can be defined and configured in the MXCS subsystem. A data source can be shared by JDBC Type 4 clients and ODBC clients.

Although MXCS provides a default data source, defined data sources offer many advantages because this allows system administrators to coordinate and manage user needs with an appropriately defined data source. An MXCS data source represents a pool of server (process) instances that share the same attributes and NonStop SQL context. These include execution priority, isolation level (for example, read only access), catalog and schema defaults, and other configurable attributes. Scalability and availability are enhanced when servers are configured to run across a range of logical CPUs on the NonStop server.

Connections can be made only to active data sources. When the T4 driver's serverDataSource property points to an active, existing data source, connections can be established to that data source. If the data source exists but is not active, access is denied. When the serverDataSource property is not specified or points to a data source that does not exist, the connection is made to the default data source, TDM_Default_DataSource, if it is active.

Figure 3. JDBC T4 driver architecture.



The T4 driver conforms, where applicable, to the Oracle JDBC 3.0 API specifications. Exceptions are noted in the Programmer's Reference manual.

References

- *JDBC Type 4 Driver Programmer's Reference for SQL/MX,* Chapter 1: Introduction to HP NonStop JDBC Type 4 Driver
- *JDBC Type 4 Driver Programmer's Reference for SQL/MX*, Chapter 4: Type 4 Driver Properties
- *JDBC Type 4 Driver Programmer's Reference for SQL/MX*, Chapter 8: Type 4 Driver Compliance
- *SQL/MX Connectivity Service Manual for SQL/MX* , Chapter 1: Overview of MXCS

Choosing a driver for Java applications running on NonStop

The T2 and T4 drivers are functionally identical. Java applications running on the NonStop server may use either of the drivers. However the T2 driver must be used to access SQL statements from within Stored Procedures in Java (SPJs), and the T4 driver must be used when the client is off-platform.

The elements described in the following sections should be considered in conjunction with careful reading of the manuals and analysis of the current and anticipated conditions that apply to the application:

Type of application
The T2 driver is appropriate for simple applications, like tools and batch processes that run single-threaded. Connection set up is easier, and, thanks to its direct interface to SQL/MX, it provides better performance.

Path length to the database.
The T2 driver uses native API calls with a Call Level Interface (CLI). This creates a shorter path to the data than is possible with the T4 driver. The T4 driver communicates over TCP/IP with a server instance of the MXCS data source, which in turn calls the CLI.

User authentication
Access to SQL (and other) objects on the NonStop server is based on operating system level user IDs. Since the T2 driver operates on-platform, the user ID of the client (the calling program) is known and valid. Connections through the T4 driver require user authentication via MXCS and the Safeguard database.

Scalability
The T2 driver and the SQL CLI are both contained within the JVM as shown previously in (see Figure 2. JDBC Type 2 driver architecture). A JDBC Type 2 call that in turn invokes the CLI to handle a SQL request will hand over work to the database engine. However, at that point, the other Java threads in the JVM will compete for CPU resources with the CLI code. T2 applications can be made to scale by adding more JVMs that process the application's requests. Applications that execute within application containers such as NSJSP and NSASJ will use multiple JVMs transparently.

Load balancing
Since the NonStop Server operating system is designed for parallel and distributed processing, load balancing is crucial for performance and effective utilization of system resources. The T4 driver might be a better choice when load balancing is a concern.

Any JDBC database connection requires memory in the calling program. This may be used for an optional SQL statement cache or, in the case of the T2 driver, the CLI library code and the memory the CLI requires for executing SQL statements. In addition, an instance of the SQL/MX compiler for each connection is required. The compiler instances all run in the same logical CPU as the SQL/MX CLI.

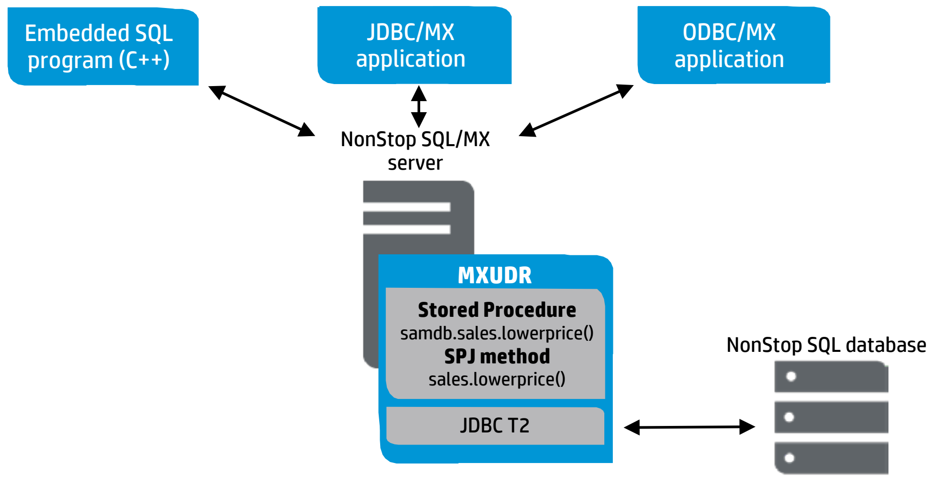
For example, a Java application that opens ten concurrent connections will start ten compiler instances in the same CPU that is executing the JVM. When the T4 driver is used, compiler instances are managed by MXCS. In this case, the ten connections will be made to ten server instances of MXCS, which can be configured across multiple processors.

Availability
Distributing an application across multiple JVMs in multiple processors increases the application's availability—a processor failure cannot take down the application completely. This approach results in a more efficient usage of the T2 driver for applications that need many connections. Using multiple JVMs reduces the number of concurrently active connections per JVM that can compete for resources in the same processor. This also demonstrates how designing for availability creates scalability.

Stored procedures in Java

NonStop SQL/MX supports Stored Procedures written as Java methods (SPJs).These methods can be coded in any Java development environment but each must be registered in the SQL/MX metadata before being invoked as a Stored Procedure. Any SQL/MX application can call an SPJ; the same SPJ method can be invoked from embedded SQL programs in C, C++, and COBOL running on the same platform and from HP ODBC/MX and JDBC/MX clients running either on- or off-platform.

Figure 4. Different applications calling the same SPJ.



NonStop SQL/MX SPJs run in a separate Java execution environment, the MX User Defined Routine (MXUDR) server process, which SQL/MX starts automatically when it encounters an application's first CALL <SPJ> statement. During its existence, an MXUDR process hosts one SPJ environment and services one or more CALL statements. When an application terminates, its MXUDR process also terminates. SPJs must use the T2 driver to access the NonStop SQL database.

References

- *SQL/MX Guide to Stored Procedures in Java*, Chapter 1: Introduction
- *Concepts of NonStop SQL/MX part 5, Stored Procedures:* HP Document ID: 4AA4-9428ENW available at hp.com/V2/GetDocument.aspx?docname=4AA4-9428ENW&cc=us&lc=en.

Run-time aspects

The use of Stored Procedures involves additional server processes (including the MXUDR process and its compiler process) to be started. This is done automatically by the SQL/MX executor; however, you may want to minimize the number of these additional processes.

Since SPJs interface with the database via the T2 driver, the tuning considerations described in the section Performance influencers apply here as well.

Applications can set certain run-time aspects prior to calling an SPJ. These run-time aspects are set using a SQL/MX-specific "Control Query Default" called UDR_JAVA_OPTIONS. If an application calls SPJs with different sets of run-time aspects, additional MXUDR instances will be started. In this case, a single server instance may have multiple instances of SPJ environments running concurrently, which is an inefficient use of system resources.

Application container JVMs running on-platform that use the T2 driver often manage multiple database connections. In this case, there will be an additional MXUDR and compiler instance for each connection. These execute in the same logical CPU as the JVM. The number of connections per JVM should be limited to prevent counterproductive resource contention, and JVM instances should be configured across CPUs for the same reason.

When an application uses ODBC or JDBC T4 client connections, database requests are executed by server instances that are configured through the SQL/MX Communications Subsystem (MXCS). The system administrator configures the number of these server instances.

When a client issues its first SQL Stored Procedure Call, an MXUDR and a compiler instance are started. If multiple applications are used, with different requirements regarding the use of Stored Procedures, these applications should not share the same MXCS server pools (or data sources).

In the SQL/MX implementation of Stored Procedures, SPJs should not call other SPJs since doing so results in the creation of an additional execution environment. Instead, SPJs should call the underlying Java methods of the stored procedure.

References

- *HP NonStop SQL/MX Guide to Stored Procedures in Java* (Chapters 1 and 2)
- *Concepts of NonStop SQL/MX part 5, Stored Procedures* HP Document ID: 4AA4-9428ENW available at hp.com/V2/GetDocument.aspx?docname=4AA4-9428ENW&cc=us&lc=en.

Integration with other (sub) systems

To access subsystems other than the NonStop SQL/MX database, NonStop Server for Java (NSJ) provides classes to interface to existing applications and to native ENSCRIBE structured files (JToolkit). The NonStop Server for Java ecosystem provides the Java Message Service (JMS) API using a NonStop specific port of Apache ActiveMQ, and uses NonStop SQL/MX as a fault-tolerant persistent message store. Java Infrastructure (JI) provides an abstraction on the Java TCP/IP sockets class to transparently use TS/MP for inter-process communication. JI mimics a normal Java socket to a client-server application pair but actually routes messages through the TS/MP message queue mechanism between them.

JToolkit

Jtoolkit is a collection of classes that can be used to access existing processes on NonStop and to manipulate the classic Enscribe file system. The JToolkit documentation (in the form Javadocs) can be downloaded from the HP Software depot (hp.com/go/softwaredepot).

References

- *JToolkit and JDBC API reference,* downloadable from the HP Software Depot at hp.com/portal/swdepot/displayProductInfo.do?productNumber=NS-API

Enscribe API for Java

This API allows a Java program to access the NonStop Enscribe File System. The Enscribe file system predates the NonStop SQL database and supports file structures such as key-sequences, entry-sequences, relative, unstructured and queue. Queue files are not supported by the Enscribe API for Java. The API supports most if not all methods to allow file manipulation - creation, read and write.

Pathway API for Java

The Pathway API for Java provides access to the logical $RECEIVE file, the common input queue for TS/MP (Pathway) server classes. This access enables a Java process to act as a Pathway server. Pathway server programs read requests from client (requester) programs and act on those requests. Java programs written to act as a Pathway server class, or as a standalone server process not managed by TS/MP, may use the Pathway API to read and reply to requests from clients written in any language.

Pathsend API for Java

TS/MP clients send requests to servers by referencing a server class, instead of a specific process or IP address and port. They do this via a Pathsend call. The TS/MP subsystem will route the request to a free server, which may run on the same processor, the same node or even on a different node in the cluster. The messages can optionally be part of a business transaction. The Pathsend API for Java contains classes to create sessions, send context-free request and reply messages or dialogs, which are multiple requests to the same server instance.

DDL2Java

The DDL2Java tool generates and optionally compiles Java source files for entries in a NonStop Data Definition Language (DDL) dictionary file. These dictionaries are used to describe the record layouts of Enscribe files and the message layouts of the messages used in Pathway client-server interactions.

NonStop Message Queue

NonStop Message Queue (NSMQ) is a JMS 1.1 compliant messaging system and is a port of Apache ActiveMQ for the NonStop platform. At the time of this document's publication, NSMQ was based on ActiveMQ version 5.9.1. The readers are advised to refer to the latest version of *NonStop Message Queue User Guide* to know the current version of ActiveMQ that the NSMQ is based on. It is implemented using technologies such as TS/MP for process management and the SQL/MX database for persisting messages. Communicating processes use the TCP/IP protocol, but the NonStop Cluster IO Protocols (CIP) subsystem allows all NSMQ brokers to listen to the same IP address and port. These brokers are messaging agents that manage the exchange of messages between messaging clients or communication with other brokers.

NSMQ can also be integrated with other Java products such as NonStop Application Server for Java (NSASJ). Indeed NSMQ and NSASJ have been integrated and tested in our labs.

Features of NSMQ

NSMQ includes the following features:

- Clustering – mechanism for configuring multiple brokers to form a cluster. In such an environment, all brokers are networked and if a broker fails, the load is distributed among the remaining brokers in the cluster. In clustering, multicasting and dynamic discovery is not supported.

- Client API support – Only JMS clients are supported. C++, .NET clients are not yet supported and is a candidate feature for a future release of NSMQ.
- Persistence – Messages can be optionally stored in an SQL/MX database.
- Destinations – Specifies the destinations that must be created when a broker starts. The following destinations are supported:
  - Queue – used for Point-To-Point messaging in first-in first-out order. Messages are consumed from the queue in the order in which they are received.
  - Topic – used for Publish and Subscribe (Pub/Sub) messaging. The message producer is referred to as the Publisher and the message consumer is referred to as the Subscriber. However, the durable subscription of messages for topics is not supported in NSMQ.
  - Composite destination – provides a mechanism for producers to send the same message to multiple destinations at the same time.
  - Virtual destination – provides a mechanism for publishers to broadcast messages through a topic to a pool of receivers who are subscribing through queues.
  - Wildcard – provides a mechanism for consumers to subscribe to multiple destinations at the same time.

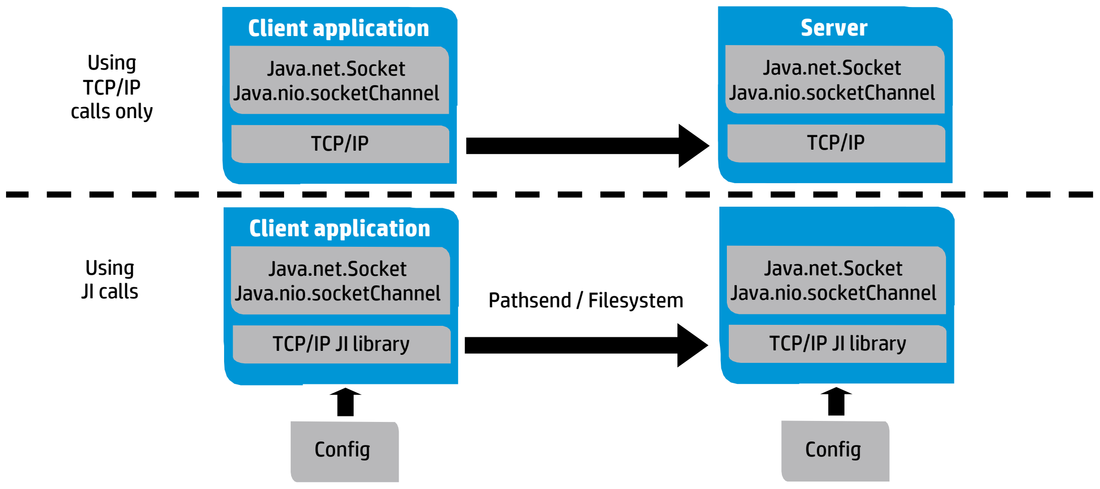
References

- *NonStop Message Queue User Guide*, Chapter 1: Introduction to NSMQ

Java infrastructure - Using TS/MP with Socket or SocketChannel

Java programs typically communicate with the outside world using the Socketclass in the java.net package or the SocketChannel in the java.nio package, both supported from NSJ7 onwards. NSJ7 has an additional feature called Java Infrastructure (JI) that allows these classes to transparently use the TS/MP infrastructure and the NonStop message system instead of TCP/IP as the underlying transport. This allows code that is designed to run on other platforms to run on NonStop, hosted within the TS/MP infrastructure, inheriting the unique features and architectural qualities on the NonStop Server. JI allows programs to be developed and tested in a workstation environment and deployed on NonStop TS/MP where it inherits the features of that middleware without the need to use NonStop Server specific proprietary API calls. JI decides, based on the contents of a mapping configuration file, if Socket or SocketChannel calls will use the standard JVM sockets implementation or if the calls are transparently changed to use native NonStop API calls. Figure 5 depicts how the JI library can replace TCP/IP calls with calls to the TS/MP server classes (Pathsend calls) or calls to a specific NonStop process using file system calls. Note that the JI library *does not replace* TCP/IP but provides additional functionality to it. This allows an application to replace only some TCP/IP calls. The affected ports are configured in the JI configuration files.

Figure 5. Replacing TCP calls with JI.



Enabling JI

The JI functionality is enabled by setting an environment variable, JI_ENABLE, to true before starting the JVM. The location of the mapping file is passed by either a system property or via another environment variable. More detailed information and examples are in the NSJ7 Programmer's Reference manual.

References

- *NonStop Server for Java 7.0 Programmer's Reference*, Chapter 5: Java Infrastructure.

Development and deployment

Development and deployment are two phases that happen repeatedly during the application lifecycle. For most applications, the development and deployment phases (including debugging) occur many times during the application's lifetime, for fixes and new versions.

Development usually takes place on workstations and not on the NonStop server. There is no specific development platform requirement for developing Java applications to be deployed on NonStop.

The dominant Integrated Development Environment (IDE) for Java is Eclipse. Eclipse can be briefly described as an extensible development platform, runtimes and frameworks for building, deploying and managing software across the entire software lifecycle. It can be used for developing in Java, but also for developing in other languages. HP provides the NonStop Enterprise Development Environment (NSDEE) which includes Eclipse plug-ins and cross-compilers specific for NonStop servers.

The method to deploy a Java application depends on the container that hosts the application, and it may be as simple as transferring an application *jar* to the correct location on the NonStop server. Refer to the specific framework manual for NonStop to retrieve specific deployment information for each application.

Debugging Java applications

NSJ supports the Java Process Debug Architecture (JPDA). Eclipse also supports Java Platform Debugger Architecture (JPDA) and can be used to debug Java programs on the NonStop server. From within NSDEE, you can control the execution of your Java program running on the host. You can set breakpoints, examine objects, set watchpoints, view threads, and perform many other operations. Stand-alone Java applications, Java applications running as server classes, Java servlets running in NSJSP, and EJBs running in NSASJ can all be debugged using Eclipse running on the desktop.

References

- eclipse.org/home/newcomers.php
- *NonStop Servlets for JavaServer Pages (NSJSP) 7.0 System Administrator's Guide*
- *NonStop Application Server for Java User Guide*
- *Open Source Java Frameworks on NonStop User's Guide*
- *NonStop Development Environment for Eclipse 4.0 User Guide*

Performance influencers

Many factors influence the performance of a Java application. Internally, the application code that runs within the JVM can cause suboptimal performance. External factors, such as database design, database hot-spots or simple system over-utilization, can also adversely affect application performance. This section specifically addresses tuning of the JVM itself, tuning of the database drivers, and the NonStop system-wide performance monitor called MEASURE.

JVM tuning considerations

Although JVM performance is heavily influenced by the application code being executed, tuning also involves configuration or environmental factors. This section describes environmental factors that most frequently affect JVM performance and tools that help tune TCP/IP configurations and examine JVM execution.

Garbage collection

JVM tuning, regardless of implementation, often involves tuning the garbage collector (GC). Although the algorithms and modeling in the NonStop Server for Java are the same as those used for Oracle JVMs, some of the information provided by Oracle may not apply to NSJ.

HP currently supports two NSJ releases. NSJ6, based on Java SE 6, does not offer parallel garbage collection. NSJ7, based on Java SE 7, does support parallel garbage collection on multi-core NonStop servers running the J-series OS versions. The JVM itself and the garbage collector(s) run in multiple cores of the logical processor, forming a process group. The 64-bit version of NSJ7 allows much more memory for Java objects. Since more garbage can accumulate, garbage collection processing can take longer and hence the heap size configured for NSJ based applications must be carefully managed.

Scale out

Scaling out with multiple JVMs may be more appropriate on NonStop than scaling up to more concurrent threads per JVM. All user threads of a JVM run in the same core of the CPU (in NSJ7) or in the same CPU (NSJ6). At some point, scaling up (with additional threads per JVM) will exceed the number of threads (and other processes) that a single processor can accommodate. The large memory model of the 64-bit version of NSJ7 makes this especially tempting, but contention among threads must always be considered when performance problems occur.

Insufficient memory and QIO segment
NSJ6 and the 32-bit version of NSJ7 may run out of memory unless the QIO segment is moved to the KSEG2 area of memory. (The QIO segment is used by the parallel TCP/IP software stack.) By default, KSEG1 contains both the QIO memory segment and user application accessible memory segments. Moving the QIO segment to KSEG2 makes more 32-bit addressable memory available to user processes. Details can be found in the Java and QIO Configuration manuals listed below.

TCP/IP configuration and javachk
Performance issues can also be caused by an incorrect TCP/IP configuration. Both NSJ6 and NSJ7 provide a command line tool, called javachk, which presents information about appropriate environment settings and response times for commonly used TCP/IP function calls. The javachk tool and a README file can be found in the $JAVA_HOME/install directory.

HPjmeter
The HPjmeter tool monitors live applications by running an agent process that communicates with the HPjmeter console on Linux, Windows or HP-UX. Individual JVMs can be examined as they execute, in order find, for example, which methods consume the most resources.

Java Visual VM
On platforms that support the full Java GUI standard, jvisualvm.exe is found in the JVM's bin directory. Jvisualvm is used to troubleshoot applications, and monitor and improve the applications' performance. Jvisualvm allows developers to generate and analyze heap dumps, track down memory leaks, perform and monitor garbage collection, and perform lightweight memory and CPU profiling. Jvisualvm uses JMX to connect to Java applications running off platform and is often used to monitor the performance on Java applications running on the NonStop server. Plug-ins also exist that expand the functionality of jvisualvm.

References

- NonStop Server for Java 6.0 Programmer's Reference, Chapter 4, Implementation Specifics
- NonStop Server for Java 6.0 Programmer's Reference, Section "Memory Considerations: Moving QIO to KSEG2
- NonStop Server for Java 7.0 Programmer's Reference, Chapter 2, Installation and Configuration
- NonStop Server for Java 7.0 Programmer's Reference, Chapter 4, Implementation Specifics
- NonStop Server for Java 7.0 Programmer's Reference, Chapter 7, Application Tuning and Profiling
- NonStop Server for Java 7.0 Programmer's Reference, Section "Memory Considerations: Moving QIO to KSEG2
- QIO Configuration and Management Manual.
- "HPjmeter downloads and documentation" at hp.com/go/hpjmeter
- Java Visual VM at docs.oracle.com/javase/7/docs/technotes/tools/share/jvisualvm.html

JDBC driver tuning considerations

JDBC driver tuning usually involves reducing the number of SQL compilations. Query compilation can often be more expensive than execution, especially in an OLTP (Online Transaction Processing) application. Though both drivers (T2 and T4) support dynamic SQL (requiring compilation of every query before every execution) it can be avoided using the processes explained below.

Prepare/execute with parameters
Statements that are used multiple times should follow the prepare/execute design pattern and use parameters to allow for re-use of the execution plan. The following example shows the pattern and use of a parameter:

```
PreparedStatement ps;
ps = conn.prepareStatement("Select cust_name from customers where cust_ID
= ?");
ps.setInt (1, entered_cust_ID);
ps.executeQuery();
```

Prepared statement cache
Statements are prepared against an active connection. Both SQL/MX drivers allow definition of a prepared statement cache which is allocated per database connection. When this is in effect, multiple compilations do not occur. Instead, the previously compiled plan for the statement is passed to the java method. For example, if the code in the previous example is executed

multiple times, the prepare statement call will only invoke the compiler once. When the connection is closed, the statement cache is released, unless connection pooling is also used.

Connection pooling

The SQL/MX drivers support connection pooling, which extends the benefits of statement caching to new connections. Without connection pooling, the driver responds to an application's close request by physically closing the connection and effectively deleting its prepared statement cache. When connection pooling is used, the driver instead adds the connection—with its statement cache intact—to a pool of free, reusable connections. The drivers dynamically adjust the size of the connection pool based on user defined maximum and minimum values. Pool size is important, because when a connection must be re-created, the statement cache must also be rebuilt.

Module File Caching (MFC)

The Module File Caching (MFC) facility provides a system-wide statement cache for the Java drivers and the ODBC/MX for Windows driver. When MFC is in place, every compiled plan is stored on disk, ready for other connections to read instead of calling the compiler.

The connection's statement cache continues to be available and will be searched first when a SQL statement is encountered. If the plan does not exist there, the system-wide module file cache will be searched. The compiler is called only when both searches fail. The new plan is placed in both the connection's statement cache and the system-wide Module File Cache.

MFC is enabled for the T2 driver with a property and for the T4 driver with a definition in MXCS.

Table 1. Connection pooling and MFC configuration examples.

Feature	**T2 driver**	**T4 driver**
Enable statement cache	jdbcmx.maxStatements=100	t4sqlmx.maxStatements=100
Set max # pooled connections	jdbcmx.maxPoolSize=10	T4sqlmx.maxPoolSize=10

Enable MFC	T2 driver properties:
	jdbcmx.enableMFC=ON
	jdbcmx.compiledModuleLocation=<dir>
		MXCS configuration:
		statement_module_caching=ON
		Compiled_module_location=<dir>

References

- Module File Cache for NonStop SQL/MX, Technical White Paper
- *JDBC Type 2 Driver Programmer's Reference for SQL/MX Release 3.2.1*, Chapter 2: Accessing SQL Databases with SQL/MX: JDBC/MX Properties
- *JDBC Type 4 Driver Programmer's Reference for SQL/MX Release 3.2.1*, Chapter 4: Type 4 Driver Properties

NonStop system tuning using MEASURE

Unlike JVM-specific performance monitoring tools (like HPjmeter and Java Visual VM) that "look inside" the JVM, the Measure performance monitor looks at system and network level resource usage. Measure data can be used for system tuning, bottleneck detection, workload balancing, and capacity planning. Multiple, concurrent Measure sessions can be configured and run independently by one or more users; this can be beneficial in a development environment.

For applications written in Java, Measure can report statistics about a JVM's resource usage as it executes its application code. Table 2 presents commonly found patterns of Measure data and suggested remedial actions that target their usual causes.

Table 2. Common patterns of MEASURE data.

Measure indication	**In MEASURE Entity**	**Remedial action**
High CPU utilization	CPU	Examine the PROCESS entities of that CPU to find the expensive processes.
High CPU utilization	PROCESS (mxcmp)	This indicates a high level of query compilation. Identify the cause(s) and amend. • Verify prepare/execute pattern is used where possible. • Verify that JDBC statement caching is enabled. • Ensure that the client JVM's cache size is large enough.
High CPU utilization	PROCESS (JVM)	Investigate these possible causes: • Too many concurrent sessions in the JVM • Excessive I/O (and its causes) • Suboptimal Java business code
High number of Launches	PROCESS (JVM)	The JVM should only launch instances of mxcmp and MXUDR. • For mxcmp, verify that connection pooling is enabled. • For MXUDR, configure SPJs appropriately.
SQL tables: High number of records accessed with low number of records used	FILE	Reading unnecessary rows "wastes" resources. Examine the SQL execution plans to find why unneeded rows are being read.
High access to non-SQL files	FILE	This may indicate repeated access to non-SQL data, such as jar files and log files. Ensure that non-SQL data are accessed efficiently and only as necessary.
High amount of physical I/O	DISK	Ensure that the Disk Access Manager (DAM) processes have sufficient cache space. Ensure that tables are partitioned so that data access is distributed across the partitions.

References
- *Measure User's Guide,* Chapter 1: Introduction to Measure

Conclusion

What started as a Java Virtual Machine has evolved into an ecosystem around NonStop Server for Java. Such an ecosystem has the benefits of opening up more standard ways to develop in Java on the one hand, but it requires more knowledge to integrate all this into performance and scalable applications. This whitepaper has summarized the components of this ecosystem and how they are integrated with the core NonStop software products such as NonStop SQL, NonStop Transaction Services (TS/MP) and the Transaction Monitoring Facility (TMF).

This paper has provided only an overview—a basic guided tour through the NonStop server for Java ecosystem—and hopefully starts your productive journey into the future.

Resources, contacts, or additional links

HP NonStop information
hp.com/go/nonstop

HP NonStop documentation
hp.com/go/nonstop-docs

HP technical white papers
hp.com/servers/technology

Sign up for updates
hp.com/go/getupdated

Share with colleagues

Rate this document

4AA5-6982ENW, February 2015, Rev. 1