# PROGRESS APPLICATION SERVER FOR OPENEDGE: TUNING GUIDE

Author: Michael Jacobs
Senior Software Architect, Progress OpenEdge

# Table of Contents

**1 INTRODUCTION.....3**

**2 SERVER ARCHITECTURE.....4**

2.1 PAS for OpenEdge and OS Process Limits.....4

2.2 Java Virtual Machine (JVM).....5
- 2.2.1 Monitoring the JVM.....5
- 2.2.2 JVM Memory Management.....6
- 2.2.3 JVM Class Loaders.....7

2.3 Common PAS Web Server.....8
- 2.3.1 PAS Memory Usage.....9

2.4 PAS Startup Time.....10
- 2.4.1 HTTP Client Connections and HTTP Requests.....10
- 2.4.2 HTTP Request Execution.....11

2.5 OpenEdge ABL Web Application.....12
- 2.5.1 Running OpenEdge Business Logic in a Multi-Session Agent Process.....13
- 2.5.2 Monitoring OpenEdge Web Applications and Multi-Session Agents.....14

**3 TUNING PAS FOR OPENEDGE.....4**

3.1 Tuning Goals and Common Steps.....15

3.2 Tuning the PAS Server.....16
- 3.2.1 Tuning the JVM Memory and Garbage Collection.....17
- 3.2.2 Tuning the PAS Client Network Connections.....19
- 3.2.3 Tuning the PAS HTTP/HTTPS Request Processing.....21
- 3.2.4 Tuning the PAS Server Features.....21

3.3 Tuning the OpenEdge Web Applications and MS-Agents.....23
- 3.3.1 OpenEdge ABL Session Pool and Request Scheduling.....26

# 1 Introduction

Progress Application Server for OpenEdge (PAS for OpenEdge) is a web server preconfigured to expose your Advanced Business Language (ABL) business and web-UI applications to HTTP clients in an internet/intranet environment. Because PAS for OpenEdge is constructed on top of a web server, it will not be configured, monitored or tuned like the classic Progress® OpenEdge® Application Server that you may be used to, nor will PAS for OpenEdge behave like it.

This document provides basic guidance for tuning PAS for OpenEdge by providing you with relevant architectural information and description of the configuration controls you will use. After reading this paper, you can expect to have a basic understanding of where to begin monitoring and tuning the execution of PAS for OpenEdge to achieve optimal performance for your ABL application. What this document will not provide is in-depth architectural and run-time information that may be required in some more advanced scaling scenarios when one PAS for OpenEdge instance has reached its maximum capacity.

The PAS for OpenEdge configuration, when tapped for development, uses a set of defaults that enable it to support a small number of web applications with moderate client loads across many platforms. Your OpenEdge business application under development runs as just another web application alongside any mobile application or third-party products you install.

The default configuration is sufficient for most development environments and some small production sites. However, that default development configuration does not guarantee that PAS for OpenEdge will operate at an optimal level for every combination of web applications. Supporting a large test or production environment requires adjustments to the default PAS for OpenEdge configuration. You can expect that you will establish a baseline PAS for OpenEdge configuration for your ABL applications and then adjust it as needed per installation based on customer hardware and usage.

As with tuning any web server environment, PAS for OpenEdge requires you to spend the time to execute a monitor-adjust-observe cycle so that you can fine-tune its configuration and attain maximum performance.

# 2 Server Architecture

Before you begin tuning your PAS for OpenEdge operation, it will help to understand the PAS for OpenEdge architecture, how your ABL application relates to it and where certain critical subsystems operate.

An ABL application running in PAS for OpenEdge is comprised of these major subsystems:

1. The OS process and its configured limits
2. The Java Virtual Machine (JVM)
3. The PAS based on Apache Tomcat
4. The OpenEdge web applications (oeabl.war)
5. The Multi-Session ABL language agent (MS-Agent)
6. The ABL application source code and architecture (ABL application)

Each of the major subsystems is described below to provide you with a basic understanding of the role it plays in tuning PAS for OpenEdge for optimal performance.

Because the JVM and Apache Tomcat are used industry-wide, there are many internet-accessible resources that provide additional tips and guidance. This document strives to use real product terms so that your internet searches will return better results. Note that to simplify local and remote administration, PAS for OpenEdge has externalized most of the Apache Tomcat server configuration options using a Java property file. Where applicable a cross reference is provided to assist in reading Apache Tomcat documentation.

## 2.1 PAS for OpenEdge and OS Process Limits

PAS for OpenEdge and any of its spawned subcomponents run in OS processes and therefore are affected by the imposed resource maximums for memory, open files and so forth. Many times, process limits differ between OS vendors, so being aware of what these limits are in the OS you are running will assist you in knowing what the maximum setting can be for some aspects of a PAS for OpenEdge configuration.

Other implied OS process limits also apply, such as CPU and memory. The number and speed of the OS CPUs greatly influence the concurrent execution speed of client requests, and the memory influences the number of concurrent client requests. When web applications use

only the PAS for OpenEdge REST and/or web transports, elastic scaling of client requests is supported across multiple instances if the following conditions are met:

- All deployed OpenEdge web application security configurations use the ***BASIC*** authentication model; and

- One or more deployed OpenEdge web application security configuration use the ***HTTP FORM*** authentication model AND the server Session-Manager is configured to use a shared cache mechanism (such as a SQL database amongst other possibilities) to share HTTP sessions across all running instances.

All other PAS for OpenEdge application configurations require the elastic load balancer configuration to use ***sticky sessions***.

## 2.2 Java Virtual Machine (JVM)

The PAS server, in which OpenEdge web applications are deployed, runs in a JVM. The JVM supplies all of the memory management, threads and I/O resources. The JVM is the point at which you make critical tuning decisions that affect every aspect of server performance, such as determining which web applications you deploy, how many clients it will support and how fast it processes those client requests.

### 2.2.1 Monitoring the JVM

External products exist that allow you to connect to a JVM and sample its operating metrics. The most basic of these products is JConsole, which is available as part of the standard Java distribution. The JVM itself has some logging capabilities that allow a level of debugging such as memory management. These logs may also be helpful in determining whether to make changes to the JVM configuration.

**Java Distribution Tools**

| JConsole | Offers a JMX-compliant graphical tool for monitoring a JVM |
|---|---|
| jvisualvm | Provides memory and CPU profiling, heap dump analysis, memory leak detection, access to MBeans and garbage collection |
| jps | Lists instrumented HotSpot Java Virtual Machines on a target system |
| jstat | Collects and logs performance statistics as specified by the command line options |

There are a number of third-party utilities that can be found on the internet if the tools listed above are not sufficient.

The JVM also has garbage collection options that can be set to provide you with text output on STDOUT for longer-term monitoring.

| JVM Option | Description |
|---:|---|
| -verbosegc | Enable verbose garbage collection output |
| -XX:+PrintGCDetails | Print detailed garbage collection information |
| -XX:+PrintHeapAtGC | Print before and after heap information when garbage collection is run |
| -XX:+PrintGCTimeStamps | Add timestamps to garbage collection data |
| -XX:+PrintGCDateStamps | Add date-stamps to garbage collection data |
| -Xloggc:<file> | Output garbage collection data to <file> |

**Note:** Garbage collection tuning is a highly technical topic that this paper does not explore. There are many excellent books, internet articles and Oracle documentation that provide a wealth of information on this topic.

### 2.2.2 JVM Memory Management

The JVM supplies two different memory regions that may be tuned for your specific ABL application running in an end-customer environment: a ***stack*** region and a ***heap*** region.

The JVM stack region holds local variables and method call parameters, with everything else allocated out of the heap region. Each JVM thread allocates space in the ***stack*** region. You can estimate the size to which the ***stack*** region will grow by multiplying the maximum number of JVM threads by the stack size configuration option value. Generally, you do not need to tune the JVM ***stack*** region, but it is tunable to save memory when running smaller web applications or to support very large complex web applications.

The JVM ***heap*** region is subdivided into ***generations***, which indicate the longevity of allocated objects before they are scanned and ***garbage-collected***, after which the memory is made available for reallocation. Once a JVM allocates OS process memory, it does not return the process memory—even if the Java application returns to an idle state and ***garbage-collection*** has made ***heap*** space available again. The OS process view of memory allocation indicates the

JVM peak memory load on the OS memory allocation, but it is not indicative of the application steady-state memory usage. The JVM internal memory allocation needs to be accessed using JVM monitoring tools.

Understanding the JVM ***generations*** becomes very important when configuring the amount of memory available to application run-time operations and in the amount of memory available to load and store application code. The JVM provides two separate configurations to control the amount of memory:

| Generation | Configuration Category | Description |
|---|---|---|
| Young | heap (-Xms & -Xmx) | Newly allocated memory reserved for run-time operations that may have a short lifetime |
| Old | heap (-Xms & -Xmx) | Young generation memory allocations whose lifetime has existed past a certain point and may exist for a longer time |
| Permanent | permgen (-XX:PermSize & -XX:MaxPermSize) | Permanently loaded Java class and statics that are almost never garbage-collected |

The JVM ***garbage-collector*** is a highly complex mechanism that is designed to reclaim orphaned or abandoned memory ***heap*** space allocations and make them available for use again. The JVM supplies different implementations of tunable ***garbage-collectors*** to handle different types of Java application loads, which in this case is Tomcat and its deployed web applications. Selecting the right ***garbage-collector*** implementation and tuning it is largely determined by which web applications are deployed and the amount of client request traffic.

The general rule is to minimize the allocation of ***stack*** and ***permgen*** space and give the rest of the available process quota to the ***heap***. The general rule for ***garbage-collector*** selection is to strike a balance between running infrequent huge collections that stall the PAS server and running frequent collections that deny CPU to the running web applications. Note that the balance point will be different for each combination of web applications and its client request frequency and data load.

### 2.2.3 JVM Class Loaders

A JVM uses a subsystem named a ***class-loader*** to load Java class byte code into ***permgen*** memory space and initialize that code. This relates to PAS in two ways: the Java security system (which is not discussed here) and the allocation of ***permgen*** memory space. The ***class-loader*** subsystem is really a hierarchy of individual ***class-loader*** instances. When a class reference is made and it has not already been loaded, the JVM walks its hierarchy of ***class-loaders*** to find it, from the root node to the current. In this way, classes can be shared across multiple ***class-loaders*** by loading them by a ***class-loader*** at a higher level in the hierarchy.

This in turn reduces the load on ***permgen*** space because multiple ***class-loaders*** do not load multiple copies of the same class into ***permgen*** space.

PAS for OpenEdge (based on Apache Tomcat) implements a hierarchy of ***class-loaders***:

1. JVM
2. Apache Tomcat **System**: Used to load specific libraries found in the bin directory
3. Apache Tomcat **Common**: Normally not used, but PAS uses it to share common libraries (common/lib) across multiple web applications to reduce ***permgen*** space allocation
4. **Web application**: Each web application instance has its own class-loader to service its WEB-INF/classes and WEB-INF/lib directories

It is important to understand how web application ***class-loaders*** can inflate ***permgen*** memory space allocation. For example, if one application is deployed five times (under different names), or five different web applications are deployed one time, and they all contain the same set of libraries and classes, the ***permgen*** space allocated is (library + class size) X 5. If those five web applications were to use the **Common *class-loader*** to load those common sets of libraries and classes, the resulting ***permgen*** space allocation would be (library + class size) X 1. The negative aspect of using a **Common *class-loader*** is that all web applications must use the compatible library and class versions. This is a common Java conundrum: Shared Java libraries for far less memory utilization and updates in a single location, versus the flexibility of mixing and matching various versions of the Java libraries.

**Note:** PAS for OpenEdge uses the **Common *class-loader*** for upward of 90% of all Java classes to support its combination of web applications and use the least amount of permgen memory space.

## 2.3 Common PAS Web Server

At the core of PAS for OpenEdge is the common Progress Application Server (PAS) platform that is used by all Progress web-based products and is capable of supporting most third-party web applications that conform to the Java Servlet 3.0 standard. PAS is an adaptation of the Apache Tomcat product that runs in a Java JVM and therefore is subject to the JVM tuning of process threads, memory allocation and file/network I/O. Therefore, PAS and any of its deployed OpenEdge and third-party web applications are configured, monitored and tuned like any ordinary Apache Tomcat web applications running in a JVM. In addition to the information found in this document, there is useful information on the internet that contains tips and instructions for tuning the Apache Tomcat server and the JVM it runs in.

Measuring PAS memory and CPU resources should begin when all of its web applications are loaded and their initial memory allocations are completed.

### 2.3.1 PAS Memory Usage

A PAS server consumes a certain amount of JVM memory according to the types of server options configured, the number of deployed web applications and the number and size of concurrent client requests.

Once a PAS instance has reached its JVM memory limits and you still need to support additional web applications or client data loads, you need to consider starting additional PAS instances and balance the deployment of additional applications and/or client load. This is an advanced tuning subject and is not covered in this document.

The following sections provide a breakout of which operations consume memory and can be manipulated to control JVM memory consumption.

#### 2.3.1.1 PAS STARTUP

PAS consumes both ***heap*** and ***permgen*** space at startup time. The number of additional server options started from within the Tomcat conf/server.xml configuration file controls ***permgen*** allocation. The size of the various server-wide object pools for threads and client request handling consumes the greatest amount of ***heap*** allocation.

#### 2.3.1.2 LOADING AND STARTING WEB APPLICATIONS

After the PAS core server is loaded and initialized, it begins the serial loading and starting of the deployed web applications. Loading and starting a web application consumes CPU, ***heap***, and ***permgen*** resources according to how the web application was implemented. Each web application loaded has a dedicated ***class-loader*** in the Tomcat ***class-loader*** hierarchy. Web application classes can be shared in ***permgen*** memory space if they are loaded by a ***class-loader*** that exists higher in the ***class-loader*** hierarchy. If a class is not loaded by a higher level ***class-loader***, it may appear in ***permgen*** memory multiple times, once for each web application that references it.

PAS does not become fully available for client access until it completes the loading and starting of all deployed web application. At the time when PAS is fully available, the web applications are still not fully loaded. Incremental loading of web application classes and memory allocations continues as client HTTP requests begin arriving for the web application to execute. It is advisable that any CPU and memory measurements begin after a warmup period of client traffic across all deployed web applications.

**Note:** PAS may fail to start due to lack of ***heap*** or ***permgen*** memory space.

## 2.4 PAS Startup Time

The time PAS requires to start and begin accepting client requests can be an issue in some environments. This can be somewhat influenced by the number and type of web applications deployed and by the individual web application configuration.

PAS loads web applications serially. When each web application begins loading, you can control how many of the application servlets are loaded and started immediately rather than waiting until the first client request is executed. Fewer servlets loaded and initialized at load time means faster PAS loading of the next web application and faster time to complete the startup sequence. This is balanced against the time required to service the initial client requests. What is not loaded at web application startup is loaded on the first client request.

The servlet configuration loading can be controlled by the servlet load-on-startup element. The default is to load when the first client requests arrive, which significantly slows down the response time for the first client request after PAS has started. You can optionally have the servlet loaded and started during PAS startup so that the first client request only has to load the additional classes not accessed during web application load. This tactic speeds up the initial client response time but causes PAS to have a longer startup time.

### 2.4.1 HTTP Client Connections and HTTP Requests

PAS manages all client network connections as HTTP/S protocol connections. Managing client network connections includes handling:

- TCP/IP connections
- TCP/IP disconnections
- Message timeouts
- Terminating infrequently used client connections to enable servicing new clients

PAS initializes and maintains two resource pools to manage the influx of client connections and HTTP requests:

- A pool of JVM threads to execute individual HTTP requests
- A pool of queued HTTP requests waiting for a JVM thread to execute in

When a HTTP client makes a socket connection, PAS waits a configurable amount of time for the HTTP request to arrive before returning an error. When the client HTTP request is received, it is either assigned directly to a JVM thread for execution or queued until a JVM thread is available.

The size of the PAS thread pool directly affects how many client HTTP requests can be executed concurrently. The size of the HTTP request queue dictates the maximum volume of client HTTP requests before PAS begins returning errors to the clients. Tuning the TCP/IP connections, HTTP request queue, and thread pools is key in managing client connections and response times. Both pools consume memory space, and the size of the thread pool controls how much ***stack*** region space is used and the size of the request queue determines how much ***heap*** region space is used.

Apache Tomcat uses the default BIO connector to service client connections and HTTP requests. A NIO connector is available, but was not considered stable enough at this time (OpenEdge 11.5, 11.6) to become the default connector type. The NIO can be configured in PAS manually, but its implementation requires more advanced tuning than BIO and can possibly reduce performance if not done correctly. Later releases of Tomcat have worked out these issues and the NIO has become the default in those releases (PAS for OpenEdge will support later Apache Tomcat releases at some point in the future).

### 2.4.2 HTTP Request Execution

When a JVM thread is available from the PAS thread pool to execute a [queued] HTTP request, a new HTTP request execution context is created and passed to the thread that then executes it. The JVM thread is bound to that HTTP request context until it either ends successfully or with a failure. Once the HTTP request is complete, the JVM thread returns to the pool to service another client HTTP request.

Unlike the classic OpenEdge AppServer, handling of HTTP requests by PAS does not inherently have the concept of “timeout” once a HTTP request begins execution. PAS does make an Apache Tomcat valve available that monitors HTTP request execution times and to log a message if any request runs over the configured amount of time.

PAS does incorporate configurable limits for how long a TCP/IP connection can be idle before a HTTP request is received and how long it can be queued before a thread from the thread pool is available. Tuning the HTTP request execution parameters affects how well PAS is able to handle client connection load and spikes in client requests.

## 2.5 OpenEdge ABL Web Application

The OpenEdge [oeabl.war] web applications that run in PAS are standard Java web applications that execute in the context of a HTTP request executing a PAS thread. Those web applications bridge the gap between HTTP requests and ABL requests. Like all web applications, each client HTTP request executes in a different thread from the PAS thread pool. ABL web applications are then tuned for CPU, ***heap*** and ***permgen*** resources like any other web application based on the level of client request activity and the data payload size.

A single ABL application installed into PAS for OpenEdge has a pool of ABL sessions shared by one or more **oeabl** web applications to execute client requests. **Oeabl** web applications are optimized so that only one instance of a class is loaded into ***permgen*** memory space at any one time. So no matter the number of ABL applications and how many ABL web applications are mapped to them, ***permgen*** space allocation remains as if only one web application were deployed.

Each **oeabl** web application hosts one or more of the following protocol transports, which can be independently enabled/disabled:

- **APSV** HTTP tunneling of the OpenEdge Application server wire protocol (provided via an **AIA** in a classic OpenEdge AppServer architecture)
- **SOAP** transport (provided via a WSA in a classic OpenEdge AppServer architecture)
- **REST** transport (provided via a REST Adapter in a classic OpenEdge AppServer architecture)
- **WEB** transport (provided via WebSpeed in a classic OpenEdge AppServer architecture)

The types of protocol transports enabled affects CPU and ***heap*** memory usage. Some transports like SOAP require a lot of CPU and ***heap*** space to process each client HTTP request, where other transports such as APSV consume far less. The REST transport lies somewhere between APSV and SOAP with regards to CPU and heap memory consumption. The level of activity in the transports and the designs and implementation of the ABL application determine how much is consumed. Do not expect a formula that can predict how much OS CPU and memory to provide—you must execute your ABL application and measure its impact on PAS for OpenEdge.

Each web application employs a common ABL Session Manager to manage a pool of ABL sessions. The Session Manager ABL session pool acts as a buffering and dispatching mechanism for handling incoming client requests. The size of the pool dynamically scales up and down according to the client load and its configuration settings, dynamically increasing and decreasing the amount of ***heap*** space used. The ABL session pool queues client requests for a certain period of time when the incoming client load outpaces the ability of the ABL sessions to handle them. If a client request exists in the ABL session pool for too long, it is canceled and an error is returned to the client. When both the ABL session pool is at its maximum and the queue space is full, an error is immediately returned to the client.

An ABL application session pool manages a pool of one or more Multi-Session Agent (MS-Agent) OS processes that hold the physical ABL sessions where client requests are executed. One MS-Agent process is always created at PAS for OpenEdge startup, with additional MS-Agent processes added if/when more ABL sessions are required to service client requests. The recommendation is to maximize the use of a single MS-Agent and only scale to more than one if necessary.

The ABL session pool maintains a configurable number of local socket connections to each MS-Agent process. The number of local socket connections dictates the maximum number of ABL requests any one MS-Agent can execute concurrently.

You can configure the ABL session pool parameters to control its total size, the number of queued client requests and the number of concurrent client requests that can be executed.

There is a side effect of the Session Manager queuing client requests and the ABL application request execution times: PAS thread pool exhaustion and rejection of client requests. Both of these contribute to longer running client requests, meaning that a PAS thread is bound to the client request longer and is unable to be reused to handle other client requests. This requires a balance of managing the PAS thread pool, versus the Session Manager ABL session pool, versus the design and implementation of the ABL application.

### 2.5.1 Running OpenEdge Business Logic in a Multi-Session Agent Process

Because the ABL language cannot physically execute in the PAS JVM process, the physical ABL sessions that execute client requests are hosted in external MS-Agents. A MS-Agent process is constructed to host multiple ABL sessions that use a threaded architecture to provide concurrent (single threaded) client request execution.

There is no relationship between the number of ABL sessions in a MS-Agent and the number of client requests it can execute concurrently. Each ABL session has a small amount of memory for language engine state and data storage, but that amount is relatively insignificant. The maximum number of ABL sessions any single MS-Agent can host is closely related to the memory/file/network resource consumption of the ***ABL application*** that runs in them. The memory footprint of any single ABL session is determined by the summation of the ABL local variables, global variables, temp-tables, number of database connections and the amount of r-code loaded.

The number of concurrent client requests any one MS-Agent can execute concurrently is implied by, but not physically tied to, the OS process threads used to execute them. Rather, the maximum number of concurrently executing client requests is based on the lesser of:

1. The maximum number of local socket connections configured to exist between the ABL Session Manager and the MS-Agent
2. The number of ABL sessions that are free [i.e. idle] to execute client requests

The relationship between Session Manager network connections and the number of physical ABL sessions in a MS-Agent is relative to the ***ABL application*** implementation. If the application is implemented to use [stateful] bound client connections the ratio of ABL sessions to local sockets is n-to-1. When the ABL application uses [stateless] unbound client connections the ratio of ABL sessions to local sockets is closer to 1-to-1.

### 2.5.2 Monitoring OpenEdge Web Applications and Multi-Session Agents

Monitoring PAS for OpenEdge involves using its extensive set of metrics gathering and query operations that are available via the PAS JMX console, and optionally via its OpenEdge remote administration web application (that is used by OpenEdge Explorer/Management). PAS for OpenEdge relies on lower overhead metrics services rather than the heavyweight logging used by a classic OpenEdge AppServer. In many cases you will not find the same logging messages in the PAS for OpenEdge log files that you found in the classic OpenEdge AppServer.

This document does not teach you how to use the PAS for OpenEdge metrics gathering and queries—that information can be obtained from the OpenEdge product documentation. This document points out general information you can gather and query for:

- Listing of sessions, clients and MS-Agent connections
- Listing of hung clients (that exceeded a specified amount of time)
- List the currently executing client ABL requests
- Reset metrics counters
- Runtime Session Manager metrics:
  - Concurrent client metrics
  - Connection timeouts
  - Request queue counts
  - Request counts
  - Read/write counts
- MS-Agent ABL session metrics, per ABL session
  - Start & end times
- APSV, REST, WEB and SOAP transport metrics
  - Request counts
  - Success/failure counts

# 3 Tuning PAS for OpenEdge

After gaining a basic understanding of the PAS for OpenEdge environment in which your ABL application will execute, you are now ready to begin the tuning process. Tuning PAS for OpenEdge begins with installing the **production** version of the PAS for OpenEdge product. The PAS for OpenEdge development product applies throttles that are not conducive to performance tuning. After the production server product is installed, follow the normal steps to create a PAS for OpenEdge instance with your ABL application installed and configured. This instance is the target of your tuning process.

**Note:** You **do not** tune the core PAS server files located in ***DLC/servers/pasoe***!

## 3.1 Tuning Goals and Common Steps

The overall goal for tuning a PAS for OpenEdge instance is to support an ABL application that meets the end customer requirements for concurrent client requests, response times and OS CPU and memory resource consumption. Achieving this goal is an iterative process starting in your ABL application development process where you determine the initial PAS for OpenEdge configuration defaults that fit your ABL application target deployment. More refined tuning then occurs at the end customer site where the OS CPU, memory and file system comes into play.

The tuning process typically follows this sequence of steps:

1. Tune the PAS server
2. Tune the OpenEdge (and any third-party web applications)
3. Tune the OpenEdge Multi-Session Agent

The following is a general set of steps to get started:

1. **Deploy the [OpenEdge] web applications** that make up your ABL application to the PAS for OpenEdge instance, increasing the ***permgen*** memory space as needed until all web applications start cleanly.

2. **Determine and set the maximum number of ABL sessions** any one MS-Agent can support based on your ABL application memory allocation of variables, r-code, buffer space, etc.

3. **Calculate the maximum ABL session pool size** where the maximum number of ABL sessions across all MS-Agents is based on the sum of:
   - Estimated maximum number of concurrently bound client application connections
   - Estimated maximum number of concurrently executing unbound client requests

4. **Set the maximum number of MS-Agents** as (***session-pool-size / maximum sessions per MS-Agent***) + 1. The +1 is for handling abnormal load spikes.

5. **Set the maximum local socket connections** any one MS-Agent can have. The goal is to maximize the number of concurrently running client requests any one MS-Agent can process. The hardware CPU resources and the level of I/O switching of the individually executing ABL requests influence the value. This number needs to be <= the maximum number of ABL sessions per agent.

6. **Set the idle resource timeouts** to lower the resource load on the server without causing MS-Agent or ABL session thrashing. These settings are directly related to how long it takes an ABL session to be started in your ABL application.

7. **Set the number of initial ABL sessions** a MS-Agent starts when it is created. This number should be kept to a minimum as it affects how long it takes the PAS for OpenEdge server to become available to handle client load. The goal is to initialize enough ABL sessions to handle the initial client load and gradually scale up (trading off initial client response times) to the PAS for OpenEdge instance full capacity.

Keep in mind that the maximum capacity of a single PAS for OpenEdge instance is finite for any given combination of ABL application and OS hardware. Once you reach a single PAS for OpenEdge instance limit within the environment it runs in, and you have still not met the end customer requirements, you are then in a position where you need to either increase the OS hardware capacity or run additional PAS for OpenEdge instances and load balance. The scaling of PAS for OpenEdge instances is not covered in this document.

## 3.2 Tuning the PAS Server

The **most important aspect of tuning your PAS for OpenEdge instance** is the JVM ***stack/heap/permgen*** memory allocations and the ***garbage-collection***. The memory configuration ultimately determines how many and what type of web applications the PAS for OpenEdge instance can have deployed, and the number of clients it can support for those web applications.

The tuning process involves an iterative process of monitoring PAS behavior for a given client and web application configuration, changing configuration values and then returning to the monitoring phase. You can expect to control the following attributes from PAS configuration:

- JVM ***stack***, ***heap*** and ***permgen*** memory allocations
- JVM garbage collection cycles and overhead
- The number of HTTP client connections
- The size of the thread pool used to execute HTTP client requests
- Client connection timeout (without a HTTP request)
- The queue size for parking HTTP client requests until a thread is available from the pool
- Socket buffer size
- HTTP compression for SOAP and REST clients
- Turning Tomcat optional features on/off
- Determining which web applications will be deployed to which PAS for OpenEdge instances

JVM configuration parameters exist in the PAS instance ***/conf/jvm.properties*** file as Java system properties. You manually edit this file to make changes.

PAS configuration parameters exist in the PAS instance ***/conf/catalina.properties*** file as Java system properties where they are easier to manage from local scripts and remote administration tools. Those Java system property values are used as variable values in the PAS ***/conf/server.xml*** file. Normally, you do not edit the PAS ***/conf/server.xml*** file as it may cause the server to not start. We recommend that you use the ***/bin/tcman.{bat|sh}*** command line utility or a text editor to maintain the ***/conf/catalina.properties*** file.

The Java distribution, commercial tools or open-source products can be used to monitor PAS resources and operations. There are a number of these products available that provide excellent viewing of the web servers behavior.

### 3.2.1 Tuning the JVM Memory and Garbage Collection

PAS process memory and garbage collection are a critical point in your tuning strategy. PAS (Apache Tomcat) reacts badly when either the ***heap*** or ***permgen*** memory regions run low on space. To keep memory available to the JVM, its ***garbage-collector*** needs to run periodically to reclaim unused space. Every time the garbage collector runs it stops all of the other JVM threads, such as client HTTP request execution threads. If the ***garbage-collector*** runs too infrequently, large amounts of unusable memory may accumulate and the collection process will take a long time resulting in spikes in client response times. If the ***garbage-collection*** process runs too often you lose CPU cycles while checking the entire memory space and you will see client response times elongate due to constant CPU drain by ***garbage-collection*** cycles.

The goal is to ensure PAS for OpenEdge has sufficient ***heap*** and ***permgen*** space allocated to handle the loading of web applications, web application memory allocations, concurrent client requests, and to reduce ***garbage-collection*** processing time.

The things that influence memory consumption are:

- The number and type of web applications deployed
- The size of the HTTP request thread pool
- The number of concurrent requests and the data size of those requests

The relevant JVM properties configured in the /conf/jvm.properties file are:

| Property | Default | Description |
|---|---|---|
| -Xms<size> | 512m | Initial Java heap size |
| -Xmx<size> | 1024m | Maximum Java heap size |
| -Xss<size> | 1024k | Java thread stack size |
| -XX:PermSize | 64m | Initial permgen memory size |
| -XX:MaxPermSize | 128m | Maximum permgen memory size |
| -XX:NewSize | 64m | Initial space used for short duration objects and indirectly how often garbage collection runs |
| -XX:MaxNewSize | 128m | Maximum space used for short duration objects |
| -XX+DisableExplicitGC | — | Disable explicit garbage collection |

**Tuning tips:**

- **Reduce the frequency** of ***garbage collection*** by starting the JVM with a larger maximum heap memory space (-Xmx)
- **Reduce repeated reallocation** of ***heap*** memory by setting the initial ***heap*** memory space equal to the maximum amount (-Xms == -Xmx)
- **Reduce repeated reallocation** of ***permgen*** memory by setting the initial size equal to the maximum size (-XXPermSize == -XX:MaxPermSize)
- **Carefully lower the JVM stack size** (-Xss) to save process memory for ***heap*** and ***permgen*** allocations
- **Increase *permgen* space** when PAS stops due to an out-of-permgen-space error

The default PAS for OpenEdge configuration provides a reasonable starting point, but due to the influences of the combination of OpenEdge and third-party web applications, they can only be considered as a starting point.

### 3.2.2 Tuning the PAS Client Network Connections

Tuning the PAS client network connections involves controlling the HTTP client TCP/IP connections. Many of the network connection properties are defined as Java system properties in the PAS ***/conf/catalina.properties*** file and have ***"psc.as."*** name prefixes. These properties can be managed using the tcman command line utility (recommended) or a text editor. Each named "psc.as..." Java system property in ***/conf/catalina.properties*** is related to an Apache Tomcat a /conf/server.xml file xml element or attribute. You can find that relationship using the command "tcman.{bat|sh} help ***psc.as.xxxxxx***".

> OpenEdge recommends that you **do NOT edit the PAS instance conf/server.xml file** UNLESS an Apache Tomcat configuration attribute that is not supplied via a "psc.as" configuration property is needed. If a new Apache Tomcat configuration attribute is required, we recommend adding the attribute to the conf/server.xml file using a "psc.as." property and adding that property to the conf/catalina.properties file where it can be remotely managed by administration tools and automated scripts.

PAS client network connections are a server-level resource and exist for each open HTTP/HTTPS/AJP13 port [connector]. You cannot control client network connections per web application. Your primary goal is to ensure that PAS for OpenEdge has enough network connection capacity to handle the total client load for all deployed web applications, inclusive of OpenEdge web applications and any additional external third-party web applications.

**Tuning tips:**

- **Coordinate the maximum client connections** to be at, or larger than, the size of the PAS thread pool and the number of queued HTTP client requests.
- **Do not attempt to enable HTTP message compression** for HTTP-connected OpenEdge clients by adding its mime-type to ***psc.as.compress.types property***—it will not work.
- **The HTTP-connected OpenEdge *clients*** use HTTP POST messages with a maximum size of 8KB, so the maximum ***psc.as.msg.maxpostsize*** is not an issue. If you are using REST or SOAP clients with very large ProDataSet transfers, this property setting may become important.

The secondary goal is to modify the HTTP message-handling if the default settings do not allow very large message/response data exchange required by some web applications.

### HTTP connection properties:

| Property | Default | Description |
|---|---|---|
| psc.as.HTTP.connectiontimeout | 20,000 | The max time in milliseconds between a TCP connection and the appearance of a HTTP or HTTPS message |
| psc.as.HTTP.maxconnections | -1 | Max client connections on the HTTP network port |
| psc.as.HTTP.compress | on | Turn HTTP compression support “on” or “off” |

### HTTPS connection properties:

| Property | Default | Description |
|---|---|---|
| psc.as.HTTPs.maxconnections | -1 | Max client connections on the HTTPS network port |
| psc.as.HTTPs.compress | on | Turn HTTPS compression support “on” or “off” |

### HTTP message properties:

| Property | Default | Description |
|---|---|---|
| psc.as.msg.timeout | 10,000 | Timeout for async requests in milliseconds |
| psc.as.msg.maxpostsize | 2097152 | The maximum size of a POST HTTP message in bytes |
| psc.as.msg.socketbuffer | 9000 | The HTTP message buffer size in bytes |
| psc.as.compress.min | 2048 | The minimum message size, in bytes, enables compression for HTTP responses |
| psc.as.compress.types | text/html<br>text/xml<br>text/javascript<br>text/css | A comma separated list of which mime types can be HTTP compressed |

### 3.2.3 Tuning the PAS HTTP/HTTPS Request Processing

PAS handles all HTTP client requests, including OpenEdge, REST, SOAP and others. Tuning concurrent client request handling involves managing the PAS thread pool and the HTTP request queues. The goal is to tune the thread pool size to handle the maximum number of concurrently executing client requests across all web applications, within the bounds established by the JVM configuration.

| Property | Default | Description |
|---|---|---|
| psc.as.executor.maxthreads | 300 | Maximum number of threads that can be created to execute client requests |
| psc.as.executor.minsparethreads | 10 | The minimum number of threads retained in the pool to service client requests |
| psc.as.HTTP.maxqueuesize | 100 | Max queue size for parking HTTP connection requests until a thread from the thread pool is available |
| psc.as.HTTPs.maxqueuesize | 100 | Max queue size for parking HTTPS connection requests until a thread from the thread pool is available |

**Tuning tips:**

- **Each thread** in the pool allocates memory from the JVM ***heap*** space. Having too many threads detracts from the amount of JVM memory available to the applications and also from the ability of PAS to handle concurrent client requests.
- **Another factor** in setting the thread pool properties is PAS startup time. The larger the number of threads initially started (minsparethreads), the longer it takes PAS to start and be available for clients. Not starting enough threads, or keeping too few threads active results in irregular client response times as new threads are created and initialized before they can be used.
- **Monitoring for determining** these values is generally obtained during peak and slack client activity times. Look for client errors due to overloaded queues or thread pool exhaustion as indicators that values need to be increased. Look for irregular client response times that indicate the minimum retained thread count is not sufficient.

### 3.2.4 Tuning the PAS Server Features

PAS comes preconfigured with certain server features and HTTP request filters (referred to as ***valves***) that are executed on each client request. While the PAS defaults work for most situations, some server features and/or filters can be turned off in the right situations to adjust security or HTTP request processing time. The recommended method of controlling server options is to use the tcman command line utility:

```
tcman.{bat|sh} feature <feature-name>={on|off}
```

| Feature-name | Default | Description |
|---|---|---|
| SecurityListener | off | Require admin rights to start the server |
| HTTP | on | Accept HTTP client connections |
| HTTPS | on | Accept HTTPS client connections |
| AJP13 | off | Turn on connector from HTTPd/IIS servers |
| Cluster | off | Enable Tomcat clustering |
| RemoteHostValve | on | Filter client access by DNS host name |
| RemoteAddrValve | on | Filter client access by IP address |
| SingleSignOn | on | Enable Tomcat SSO across web applications |
| AccessLog | on | Track client HTTP requests and responses |
| CrawlerSessionManager | on | Reduce session pool impact from Web Crawler applications |
| StuckSessionValve | on | Report suspected hung HTTP requests |

**Tuning tips:**

- **The StuckSessionValve** applies to all deployed web applications, including the OpenEdge ones that execute the ABL language requests. This setting can result in false-positives, so interpret the information as informational and not an error.
- **The CrawlerSessionManager valve** protects against large numbers of HTTP sessions being started by external web page indexers, which use up a lot of heap space. For internet-facing PAS for OpenEdge instances, we recommend that you leave the feature on. If you have ***intranet*** PAS for OpenEdge instances, you may turn the feature off.
- ***The AccessLog feature*** uses processing time to format and write the tracking of HTTP clients. Its value to an enterprise is subjective and it should be turned off unless deemed necessary.
- ***The SingleSignOn (SSO) feature*** allows users to log into the PAS for OpenEdge instance one time for all web applications in a certain ***realm***. Both the Apache Tomcat management and OpenEdge remote administration web applications share the same ***realm*** and benefit from this SSO. OpenEdge web applications (oeabl.war) can also benefit from this functionality if they are configured for ***container*** security. If your production PAS for OpenEdge instance does not deploy those administration web applications or needs to benefit from SSO, disable this feature.
- **PAS for OpenEdge instance clusters** are advanced web server architectures and should be left turned off until necessary to support your ABL application using multiple PAS for OpenEdge instances.

- **The SecurityListener feature** is seldom used but is provided when needed for best-practice production security configurations. Only enable this feature when you are sure PAS for OpenEdge instances are deployed using the required OS file/directory permissions.
- **The RemoteHostValve and RemoteAddrValve default configurations** allow all host names and addresses, so their overhead is small. Both DNS and IP addresses are considered unreliable for identifying ***internet*** clients so you may turn these off if PAS for OpenEdge is internet facing. They may be of more use when PAS for OpenEdge is running on what the end user site considers a secured ***intranet***.

## 3.3 Tuning the OpenEdge Web Applications and MS-Agents

After you tune PAS for handling client requests, you can begin working with the OpenEdge web applications and MS-Agent processes. The types of information found in this section of the document includes:

- Controlling the size of the ABL Session Manager ABL session pool
- Managing resource consumption within a Multi-Session agent process
- Managing concurrent ABL request execution
- Managing PAS for OpenEdge startup time

The design and implementation of your ABL application affects the PAS and all web applications deployed in it. In most cases the ABL application design and implementation cannot, and should not, be changed. Your goal is to strike a balance between PAS for OpenEdge instance startup times, supporting client requests to your ABL business logic and the response time to those clients.

One of the realities in getting the most performance from your PAS for OpenEdge instance is realizing that ***OpenEdge web applications and the MS-Agent process execution of client requests is gated by your tuning of the PAS they operate in***. For example, it does not matter if you configure OpenEdge to handle 400 concurrent client requests if the PAS it runs in can only manage supporting 200 concurrent client requests—the net maximum client support will be 200.

Another reality is that the execution speed of the ABL language engine and the OpenEdge RDBMS storage engine is tied to the speed of the OS, the CPUs and the file system, not PAS for OpenEdge. You cannot tune PAS for OpenEdge to execute the same ABL r-code and data storage operations to run faster than a batch process or a classic OpenEdge AppServer does. It is true that shared-memory (i.e. self-service) OpenEdge RDBMS storage engine connections are much faster than those of a classic OpenEdge AppServer, but the speed increase is due to the shared-memory database connection implementation and not the database storage engine itself.

The ABL application ABL session pool is managed by the SessionManager subsystem, which is common to all of its configured OpenEdge web applications. How the SessionManager manages the ABL session pool, its pool of MS-Agents, and the local network socket connections, is controlled through properties found in the PAS for OpenEdge instance ***/conf/openedge.properties*** configuration file. This file is managed by using the PAS for OpenEdge instance ***/bin/oeprop.{bat|sh}*** command line utility, OpenEdge Explorer/Management (if the OpenEdge remote administration web application is installed), or via a simple text editor.

| Property [AppServer.SessMgr] | Default | Description (per ABL Application) |
|---|---|---|
| numInitialAgents | 1 | Number of MS-Agent processes to create at server startup |
| maxAgents | 10 | Maximum number of MS-Agent processes that can exist |
| maxABLSessionsPerAgent | 200 | Maximum number of ABL sessions per MS-Agent process |
| maxConnectionsPerAgent | 16 | Maximum number of network connections between Session Manager and an MS-Agent |
| requestWaitTimeout | 15,000 | Maximum time, in milliseconds, that a client request will be queued waiting for an ABL session before an error is returned |
| idleSessionTimeout | 300,000 | Maximum time, in seconds, that an ABL session can remain idle before it is shut down |
| idleAgentTimeout | 300,000 | Maximum time, in seconds, that a MS-Agent can remain idle before it is shut down |
| agentListenerTimeout | 300,000 | Maximum time the SessionManager will wait for an MS-Agent to report "started" before an error is raised |
| idleConnectionTimeout | 300,000 | Maximum time, in seconds, that APSV client can remain idle before it is shut down |

| Property [Appserver.Agent] | Default | Description (per MS-Agent) |
|---|---|---|
| numInitialSessions | 5 | Number of ABL sessions started at MS-Agent startup time |

**Note:** Full property descriptions are located in the PAS for OpenEdge ***instance /conf/openedge.properties.README*** file.

How the OpenEdge properties are set depends, to some extent, on the operating model implemented by your ABL application:

1. **Stateful** (i.e. classic OpenEdge AppServer state-reset/state-aware)
2. **Stateless** (i.e. classic OpenEdge AppServer stateless/state-free)

Why these application model implementations affect property settings requires an understanding of how the ABL SessionManager manages the ABL session pool, the MS-Agents where the physical ABL sessions run in and how ABL sessions are selected to execute client requests.

The first priority of the SessionManager is to conserve system resources by maximizing the pool of ABL sessions and scaling a larger number than is needed to handle the current client load. Where the classic OpenEdge AppServer used a round-robin scheduler to distribute client requests across all existing ABL sessions, PAS for OpenEdge uses a find-first-free scheduler. For example, when a classic OpenEdge AppServer handled requests from a single client, all of the running ABL sessions were used consuming OS memory and file system handles linearly. PAS for OpenEdge uses only one ABL session, consuming only one session's worth of OS memory and file system handles.

For informational purposes, the following is a general description of how the SessionManager implements its find-first-free scheduling of client requests using MS-Agents, local socket connections and ABL sessions:

1. **Receive a client request** from an oeabl web application transport.

2. **The first MS-Agent** in the list of MS-Agents is set as the current agent.

3. **If the current MS-Agent** has an unused socket and a free ABL session, use it to execute the client request:
   - During this time the local socket is placed into an ***in-use*** state and the free ABL session is ***bound***, where neither is available for use by other client requests.
   - When the client request ends, the ABL session is marked as ***free*** if a persistent procedure has not been executed and the ABL application code has not set a bound-client condition. The local socket returns to the ***unused*** state.

4. **If the number of current MS-Agent** local sockets is < max, and any ABL sessions are free, create a new socket. Go to #3.

5. **If another MS-Agent** exists in the MS-Agent list, set it as the current agent. Go to #3.

6. **If the max number of MS-Agents** has not been reached, create a new MS-Agent, add it to the end of the MS-Agents list, make the new MS-Agent the current agent and go to #3.

7. **Queue the request** if the maximum request queue size is not exceeded. Queued requests are de-queued and executed as soon as a MS-Agent socket connection becomes available.

A classic OpenEdge AppServer has four modes of operation providing two distinct application architecture models: stateful, in which one ABL session exists for each application client and stateless, in which the ABL sessions are shared by all application clients. PAS for OpenEdge provides support for both models, where all client requests are handled stateless and the ABL application can, at its direction, temporarily bind the client and operate in a stateful manner. The implications of this translate into how many ABL sessions and MS-Agents to configure.

### 3.3.1 OpenEdge ABL Session Pool and Request Scheduling

This section provides information that enables you to control PAS for OpenEdge startup times, ABL sessions for executing client requests and the concurrent client request execution.

#### PAS FOR OPENEDGE STARTUP TIME AND HANDLING INITIAL CLIENT REQUESTS

The SessionManager must start a minimum of one MS-Agent when a PAS for OpenEdge instance is started. During an MS-Agent startup process, the SessionManager stalls until it is gets a call-back connection from the SessionManager to report its startup status and any subsequent notifications. The SessionManager stalls for a **agentListenerTimeout** period before it considers the MS-Agent either unresponsive or unable to start.

PAS for OpenEdge SessionManager uses the property **numInitialAgents** to begin building a list of MS-Agent processes. The MS-Agent process uses the **numInitialSessions** property at startup time to begin building its free idle list of physical ABL sessions. For example, one MS-Agent with five initial ABL sessions yields an initial ABL session pool size of five. Starting three MS-Agents with five initial ABL sessions each yields an initial ABL session pool size of 15.

**Tuning tips:**

- **The longer the agentListenerTimeout property** is set, the longer the PAS for OpenEdge instance takes to start when for any reason the MS-Agent process has startup failures. Too short a time and the PAS for OpenEdge instance may start sooner but the SessionManager may consider the MS-Agent unavailable.
- **Setting the number of MS-Agent processes** and physical ABL sessions to start at initialization time is dependent on how long the ABL application implementation takes to run its startup procedure. A larger number takes more time, which causes the PAS for OpenEdge instance to not handle incoming client requests for a longer period of time. Too few and the first client requests handled by PAS for OpenEdge will have slow response times while new ABL sessions and/or MS-Agent processes are started.

## SCALING MS-AGENTS UP AND DOWN AND PHYSICAL ABL SESSIONS

After initial startup the SessionManager automatically scales the number of ABL sessions according to the client load. SessionManager scaling begins with adding ABL sessions to the first MS-Agent process until its **maxABLSessionsPerAgent** limit is reached. If additional ABL sessions are required to meet client demand, the number of physical ABL sessions are scaled up in the next MS-Agent process until its **maxABLSessionsPerAgent** limit is reached. This scaling process continues until the **maxAgents** limit is met, at which time the SessionManager begins to return ***no session available*** errors to its clients.

**Tuning tips:**

- **Setting the MS-Agent maximum ABL sessions** property is related to the OS process memory and/or file limits. Each ABL session that runs your ABL application code consumes memory and other process resources. Do not set the maximum ABL sessions to a higher number than the OS process supports.
- **The best performance** is achieved by keeping the number of MS-Agents to as few as possible by maximizing the number of ABL sessions and concurrent client requests per MS-Agent.
- **The more MS-Agents** that are started, the higher the overhead in dispatching incoming stateless client requests for execution. The ideal case for stateless ABL application models is one MS-Agent and a few ABL sessions.
- **Scaling to handle more MS-Agent processes** is normal when the ABL application supports stateful client requests. Overhead for stateful clients is low because they are directly dispatched for execution to an exact ABL session within an exact MS-Agent, without the need to find an idle ABL session.

As stated above, the SessionManager goal is to conserve resources. It does this by periodically scanning all of the MS-Agent and physical ABL sessions looking for ones that have been idle beyond the **idleAgentTimeout** and **idleSessionTimeout** periods respectively. When an idle MS-Agent or physical ABL session exceeds the idle time, it is removed from service and shut down gracefully.

**Tuning tips:**

- **If your client traffic peaks and valleys** often, this may lead to thrashing where client response times suffer because MS-Agents and ABL sessions are harvested too soon and the client must wait for new ones to be started up.
- **If you see from the OpenEdge metrics** too many MS-Agent/ABL session startups and idle resource shutdowns, lengthen the time they can remain idle before being harvested.
- **You also have the option of manually controlling** the harvesting of idle MS-Agents and ABL sessions. To disable the automatic idle resource harvesting, configure the **idleResourceTimeout** property to 0 (zero) and use the JMX or remote OpenEdge administration plugin to manually harvest idle resources.

## MANAGING CONCURRENT REQUEST EXECUTION

An MS-Agent can only concurrently execute the same number of client requests as it has local socket connections between the SessionManager and itself. The SessionManager automatically scales the number of local socket connections to MS-Agents up to the **maxConnectionsPerAgent** limit while looking for a free (idle) ABL session to execute a client request. A byproduct of managing the number of local socket connections is the MS-Agent management of its pool of OS threads reserved for executing client requests. Each time a new local socket connection is created, a new OS thread is started to execute client requests arriving via that socket. Each time a local socket connection is closed, an OS thread from the pool is stopped. Once a local socket connection is created and an OS thread exists to execute client requests, it can be used by any client request regardless of the bound-client state.

**Tuning tips:**

- **Do not set the maximum number of local socket connections** higher than the maximum number of physical ABL sessions.
- **Do not set the maximum number of local socket connections** higher than the maximum number of ABL sessions.
- **Do not set the sum of local socket connections** for all MS-Agents larger than the PAS thread pool for executing HTTP requests as they will never be used.
- **The choosing of an appropriate number of local socket connections** is closely tied to the implementation of the ABL application. If the ABL application uses high levels of file/database/socket I/O, you need to increase the number of socket connections and ABL sessions because many of them will have high idle times where CPU resources can be used for executing CPU-bound ABL language statements.

**Note:** The starting and stopping of local socket connections and the OS threads in its thread pool is not related to the startup/shutdown of physical ABL sessions.

With the information presented in this whitepaper, you should now have a basic understanding of where to begin monitoring and tuning the execution of PAS for OpenEdge to achieve optimal performance for your ABL application. You can expect to establish a baseline PAS for OpenEdge configuration for your ABL applications and then adjust it as needed per installation. If you have questions regarding your PAS for OpenEdge installation, please contact us.

CONTACT US

## About OpenEdge

Progress® OpenEdge® is an innovative application development platform that enables you to build and protect beautiful on-premises, cloud or hybrid applications, and deploy them across any platform or mobile device. OpenEdge meets all your digital business needs, delivering total data management for any data source with enhanced performance, minimal IT complexity and the industry's lowest total cost of ownership. Business apps built with OpenEdge are reliable, scalable and require fewer administrative resources than those built using alternative platforms.

## About Progress

Progress (NASDAQ: PRGS) is a global leader in application development, empowering the digital transformation organizations need to create and sustain engaging user experiences in today's evolving marketplace. With offerings spanning web, mobile and data for on-premises and cloud environments, Progress powers startups and industry titans worldwide, promoting success one customer at a time. Learn about Progress at www.progress.com or 1-781-280-4000.

Progress and Progress OpenEdge are trademarks or registered trademarks of Progress Software Corporation and/or one of its subsidiaries or affiliates in the U.S. and/or other countries. Any other trademarks contained herein are the property of their respective owners.

© 2016 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.
Rev 16/09 | 160917-0023

Progress®