PERCHE' GLI ENGINE VANNO IN OVERRUN

Technical Note 046 - REV.1

11/01/2024

Introduzione

Application Server è una piattaforma molto potente ed estensibile per costruire sistemi di Tecnologia Operativa (OT) moderni e dinamici.

Tuttavia, questa potenza comporta una grande responsabilità.

Un problema che si riscontra troppo spesso nei sistemi di produzione è il superamento del periodo di scansione da parte degli Engine.

Questo si manifesta tramite un rallentamento nell'aggiornamento dati Intouch/OMI.

In questo articolo esamineremo cos'è e cosa fa un Engine. Verranno inoltre analizzate le cause più comuni di sovraccarico degli Engine e come evitarle, oltre a fornire informazioni sul monitoraggio della salute degli Engine in generale.

Che cos'è un Engine?

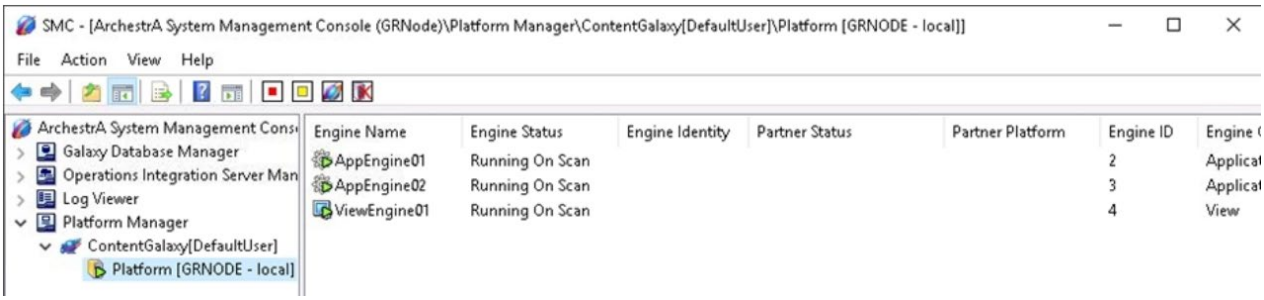
Un Engine in System Platform è un processo a 32 bit che pianifica ed esegue gli oggetti ospitati. Il processo Application Engine (aaEngine.exe) può essere uno dei seguenti:

1. Un engine integrato in un oggetto **WinPlatform**
2. Un engine standalone (**AppEngine**)
3. Un engine di visualizzazione (**ViewEngine**)

Ogni processo Engine ha un nome unico per la Galaxy e un ID Engine unico per la WinPlatform su cui viene eseguito. L'engine incorporato nell'oggetto **WinPlatform** ha sempre l'ID engine 1.

Un esempio di Galaxy che mostra i **tipi di engine**:

Per visualizzare i dettagli dell'engine, aprire la sezione **Platform Manager** dell'**SMC/OCMC**:



Poiché tutti i motori hanno lo stesso nome di processo (**aaEngine.exe**) in **Task Manager**, può essere difficile identificare quale motore sia.

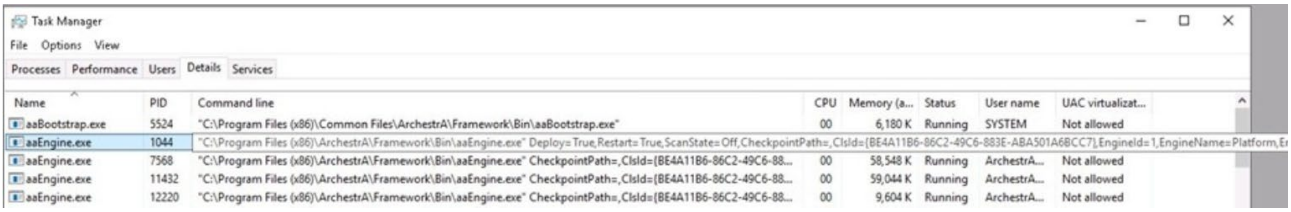
Tuttavia, è possibile visualizzare un'altra colonna nel pannello **Details** del **Task Manager** per identificare ciascun engine. A tale scopo, procedere come segue:

1. Aprire Task Manager e passare alla scheda **Details**.

2. Fare clic con il pulsante destro del mouse sull'intestazione di una colonna e premere **Select Colums**.

3. Individuare l'opzione **Command Line** e selezionarla.

4. Premere OK. La colonna **Command Line** viene visualizzata come mostrato di seguito:



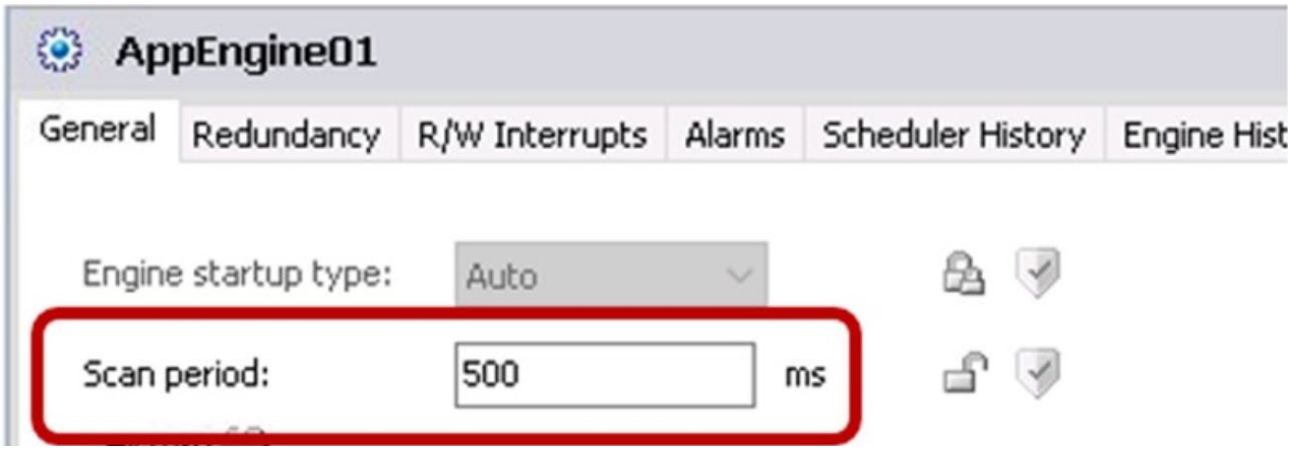
Il parametro **Engine name** identifica l'Engine, insieme all'Engine ID. Nella Figura qui sopra, è mostrato l'engine incorporato nella WinPlatform.

Comportamento Runtime dell'Engine

Scan Cycle

Gli Engine funzionano in modo ciclico. Il tempo che l'Engine ha a disposizione per completare tutto il suo lavoro in un ciclo è noto come **Scan period**.

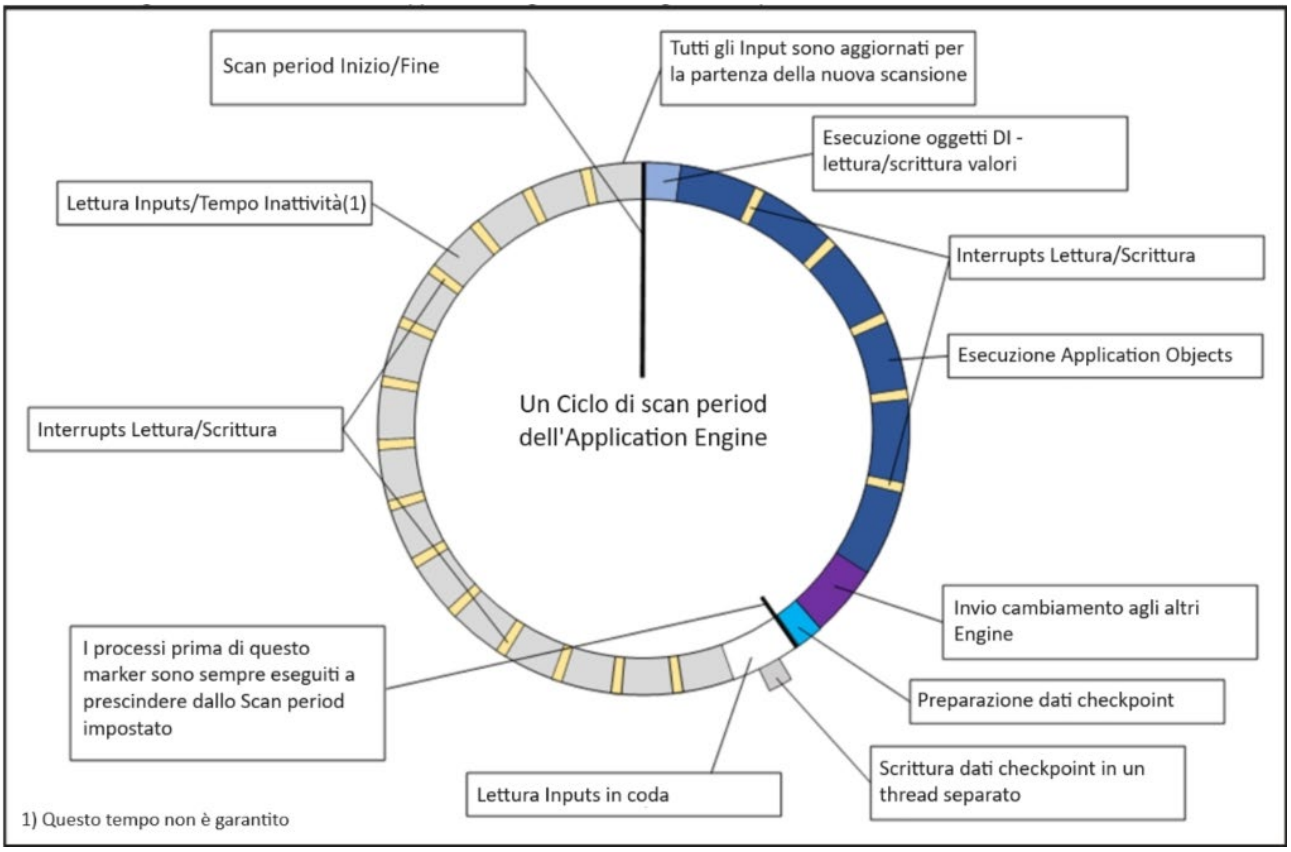
È possibile impostare Scan period individuali per ciascun Engine (**WinPlatform**, **AppEngine** o **ViewEngine**) tramite la proprietà **Scan period** (ms):



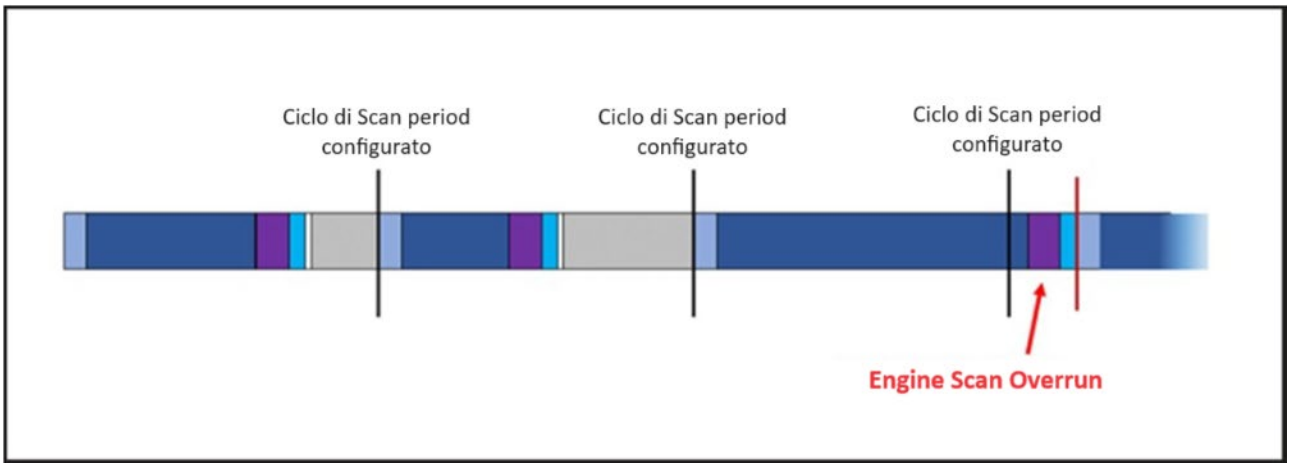
La Figura sotto illustra un normale ciclo dell'Engine. Le sezioni blu e viola rappresentano il tempo di esecuzione di tutti gli oggetti ospitati (compreso lo scripting su questi oggetti).

Il resto del tempo (rappresentato in grigio) è dedicato alla gestione delle comunicazioni dei devices (IO) e al tempo di inattività. Inoltre, c'è una piccola finestra dedicata all'aggiornamento dei dati di checkpoint (valori di runtime che vengono memorizzati su disco per essere conservati durante il deploy/failover).

Il tempo di esecuzione può variare da un ciclo dell'Engine all'altro in base al carico di lavoro degli oggetti e alle prestazioni delle risorse di sistema. In genere si consiglia di avere un tempo medio di inattività (rappresentato dall'attributo incorporato **Scheduler.TimeIdleAvg**) maggiore o uguale al 50% dello Scan period dell'Engine. In questo modo si garantisce che l'application engine abbia tempo sufficiente per elaborare in tempo tutto il carico di lavoro pianificato.



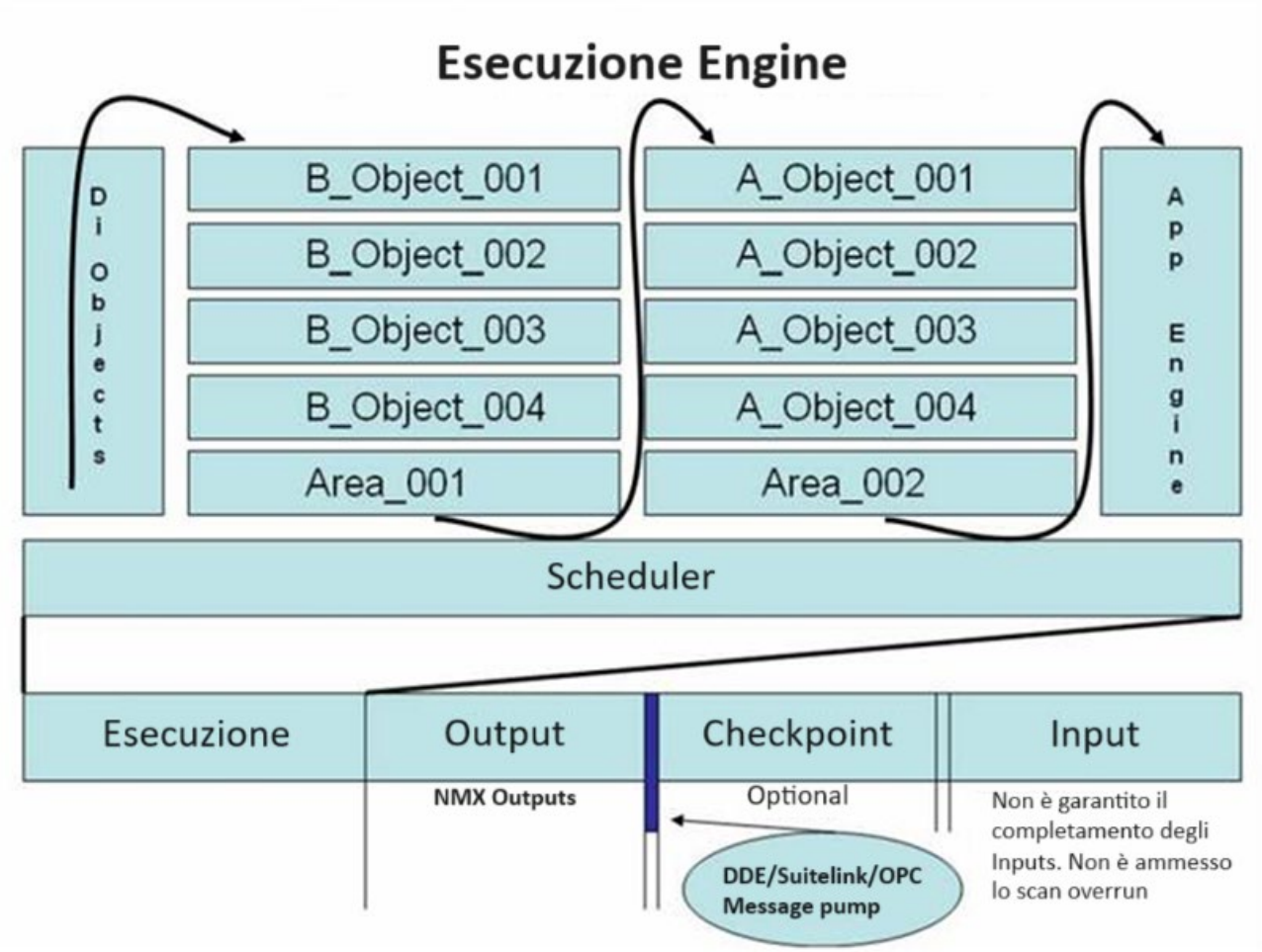
Se l'Application Engine non è in grado di completare tutto il carico di lavoro programmato entro il periodo di scansione indicato, si sfora nel periodo di scansione successivo. Questo fenomeno è noto come "scan overrun" (Figura sotto):



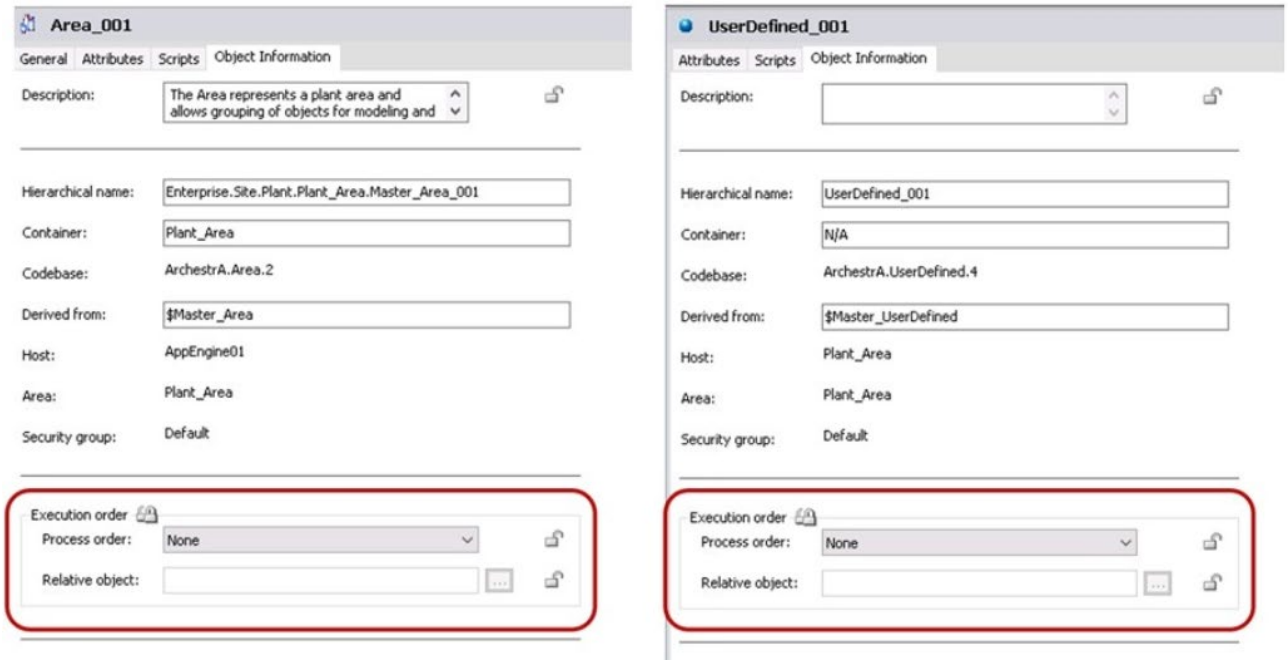
Nota: Sebbene gli oggetti **WinPlatform** e **ViewEngine** non possano ospitare direttamente altri oggetti applicativi, possono comunque contenere ed elaborare attributi di I/O e script e quindi possono andare in overrun.

Ordine di esecuzione degli oggetti

L'ordine di esecuzione di tutti gli oggetti ospitati dall'Application Engine è mostrato nella seguente Figura:



Gli oggetti vengono eseguiti in ordine alfabetico. Tuttavia, è possibile modificare l'ordine di esecuzione degli oggetti Area e degli Automation Objects configurando l'ordine di esecuzione rispetto a un altro oggetto (Figura sotto):



Non è possibile modificare l'ordine di esecuzione degli oggetti **Device Integration**, degli oggetti **Engine** (App o View) o dell'oggetto **WinPlatform**. Questi vengono sempre eseguiti in ordine alfabetico.

Principali cause dell'Engine Overrun

Le ragioni per cui un Engine può sforare sono numerose. I tre motivi principali sono:

1. Objects scripting
2. Sovraccarico
3. Ambiente

Le sezioni seguenti illustrano gli errori più comuni riscontrati in queste aree e come evitarli.

Object Scripting

L'estensibilità è una componente potente di Application Server. A ciò si aggiunge la capacità di sfruttare i.NET Framework namespaces. Qui sta il problema.

È fin troppo facile scrivere uno script che sottopone il motore a un carico inutile. Se si moltiplica questo fenomeno per centinaia o migliaia di istanze, si ottiene una ricetta per il disastro. Molti dei problemi di scripting riscontrati dall'assistenza tecnica AVEVA possono essere evitati con una progettazione ben studiata.

Non-execute Scripts

Application Server consente di scegliere in quale fase dell'esecuzione dell'oggetto può essere eseguito un determinato script. Per impostazione predefinita, la maggior parte degli script viene eseguita durante la fase **Execute** di un oggetto.

Tuttavia, se si utilizzano script non-execute, come gli script **OnStartup** o **OnScan**, è necessario stare molto attenti date le seguenti ragioni:

1. Questi script hanno una finestra di esecuzione limitata rispetto a uno script di Execute. Pertanto, possono ritardare o bloccare l'Engine durante l'esecuzione.

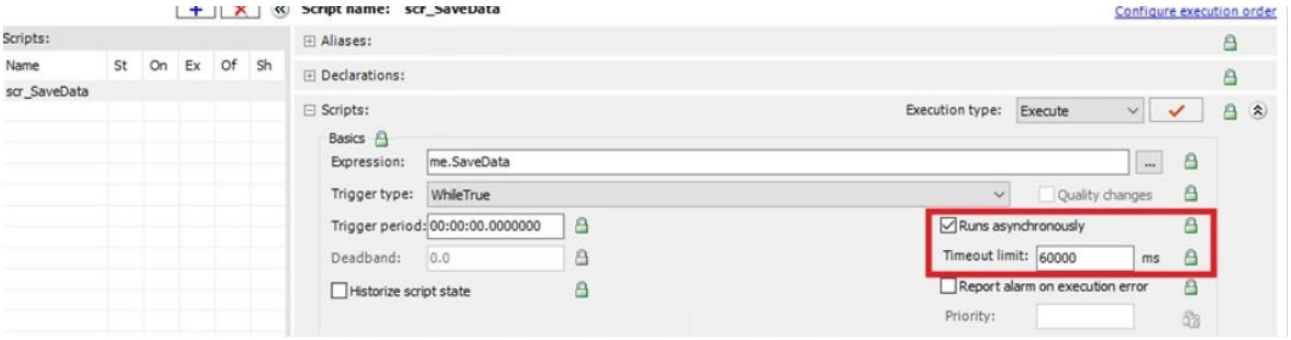
2. L'esecuzione di tale logica può allungare i tempi di deploy/undeploy e, in casi estremi, portare al fallimento dell'operazione.

3. Alcuni dati non sono disponibili per questi script. Gli attributi degli oggetti in esecuzione su altri Engine non sono disponibili per gli script **OnStartup**, ad esempio.

Questi script sono progettati per l'inizializzazione di attributi/variabili locali e devono essere veloci da eseguire. Per la logica complessa (looping, operazioni su DB, ecc) bisogna utilizzare gli script in modalità **Execute**.

Blocking scripts

Quando si scrivono script pesanti che possono bloccarsi per via dello scan period, assicurarsi che lo script sia impostato per l'esecuzione **asincrona** (Figura sotto):



Esempi tipici di script che possono bloccarsi sono gli script SQL, soprattutto quando ci si connette a un database e si legge/scrive su un file.

Queste operazioni possono facilmente superare il periodo di scansione dell'Engine, soprattutto se il periodo di scansione è impostato su un valore predefinito di **500ms**. Se lo script è in esecuzione in modo **sincrono**, l'Engine attende che lo script finisca e non continuerà l'esecuzione finché non sarà terminato. Questo avviene nel thread principale dell'Engine. Questi script possono essenzialmente bloccare l'esecuzione di qualsiasi altra cosa da parte dell'Engine e questo blocco porta **all'overrun**.

In uno script **asincrono**, l'Engine avvia un nuovo thread ed esegue lo script in quel thread. Punti da notare:

1. L'Engine non ha alcun controllo sull'esecuzione dello script in quel thread separato.

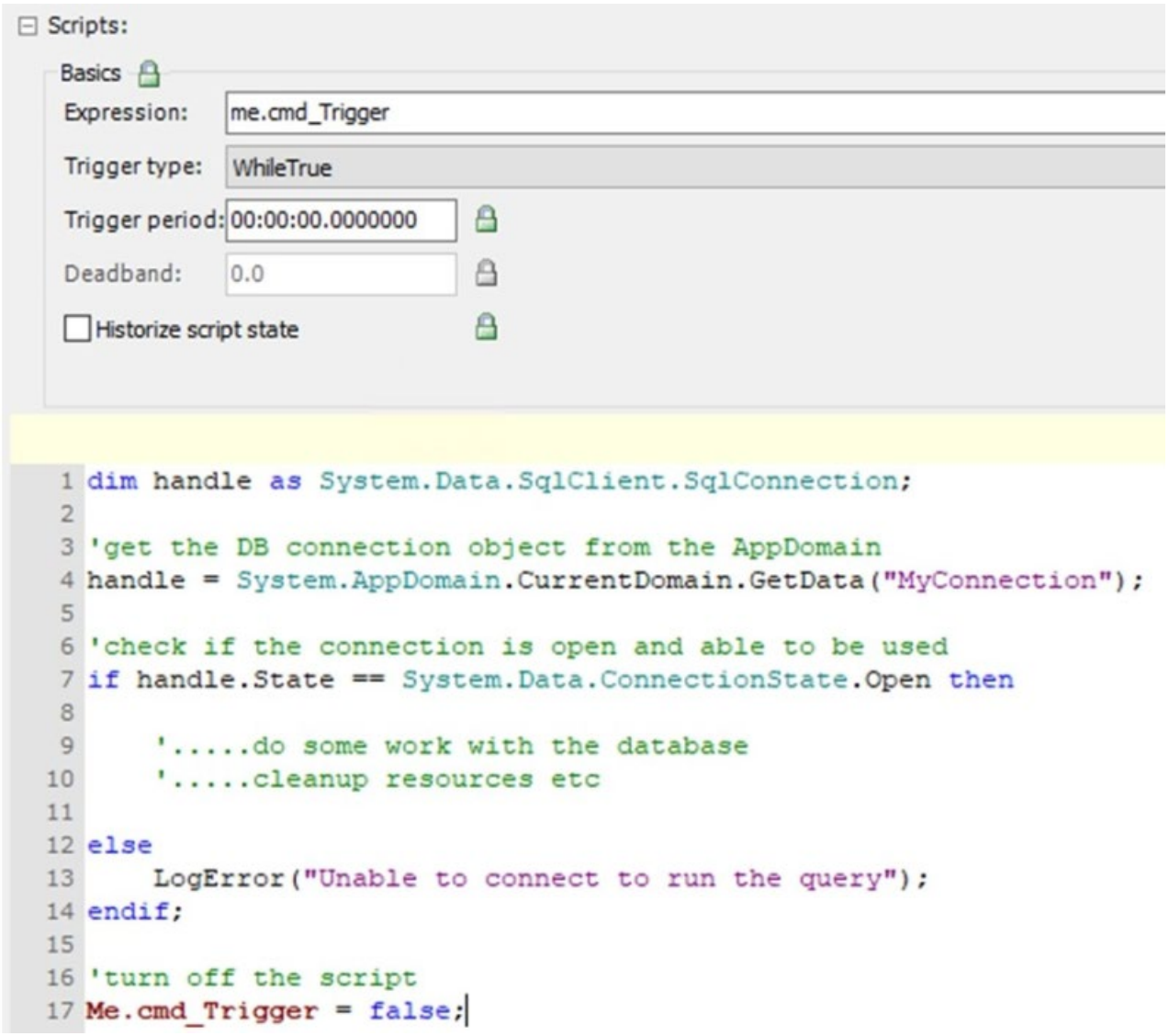
2. Quando lo script è in esecuzione, l'Engine prosegue con l'esecuzione degli altri script o dell'attività successiva; in altre parole, l'Engine non attende il completamento dello script prima di passare ad altro. **È necessario considerare la gestione di eventuali problemi di temporizzazione tra gli script di un Oggetto che potrebbero sorgere con l'uso di script asincroni.**

3. Esiste un limite al numero di script asincroni che possono essere eseguiti contemporaneamente. Questo limite è gestito dalla **Maximum Asynchronous Thread Count** nella scheda **General** della configurazione dell'**Engine**. Il numero predefinito è cinque. È possibile aumentare questo numero, ma occorre tenere presente che la creazione di thread aggiuntivi ha un costo. **Se si raggiunge il limite massimo, gli script successivi vengono messi in coda finché uno degli script in esecuzione non è terminato.**

Blocco TRY:CATCH & reset dei trigger

Tutto il codice che può lanciare un'exception deve essere racchiuso in un blocco **TRY:CATCH**. Application Server dispone di questa funzionalità da molti anni, ma molti non la usano ancora.

Si consideri il seguente esempio:

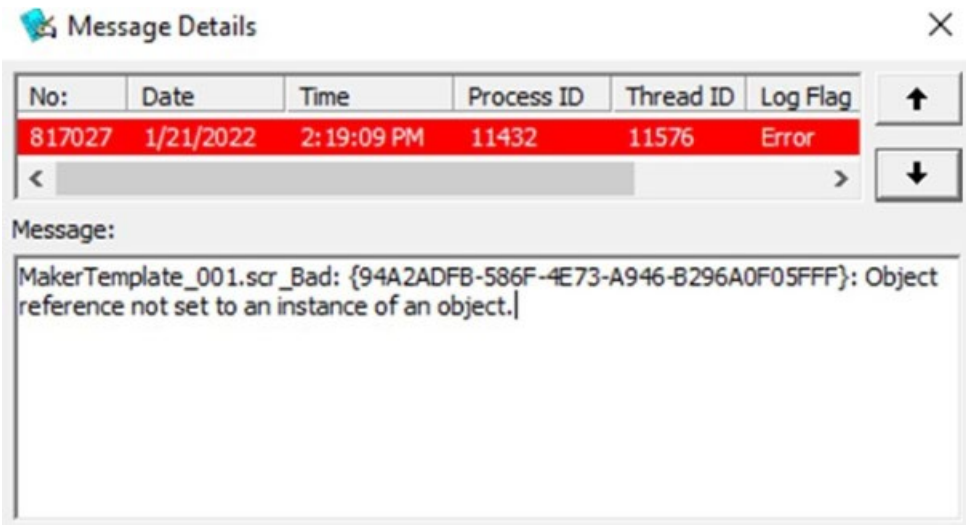



```
dim handle as System.Data.SqlClient.SqlConnection;

'get the DB connection object from the AppDomain
handle = System.AppDomain.CurrentDomain.GetData("MyConnection");

'check if the connection is open and able to be used
if handle.State == System.Data.ConnectionState.Open then

    '.....do some work with the database
    '.....cleanup resources etc

else
    LogError("Unable to connect to run the query");
endif;

'turn off the script
Me.cmd_Trigger = false;
```

Lo pseudo-codice sopra riportato è un esempio di un bug molto sottile che è facile da ignorare. In questo esempio, lo script ottiene un oggetto SQL Connection dall'**AppDomain** e poi verifica se può essere utilizzato, utilizzandolo in caso affermativo e gestendolo in caso contrario. Infine, lo script viene disattivato (il trigger viene resettato).

Tuttavia, lo script fa una supposizione fondamentale che può causare un loop: presuppone che un oggetto **SQL Connection** sia stato restituito nella chiamata al metodo **GetData()**. Se questo non è mai stato impostato, il metodo **GetData()** restituisce **NULL**. La riga di codice

successiva non effettua alcun controllo e utilizza la connessione. Tuttavia, poiché la variabile handle è **NULL**, viene lanciata una **NullReferenceException** e lo script interrompe l'esecuzione a questo punto:



Il risultato è che il trigger dello script: **cmd_Trigger** non viene mai impostato su False e quindi lo script va in loop.

Il problema è aggravato dal fatto che questo codice non è circondato da un blocco **TRY:CATCH**. Il blocco **CATCH** è il concetto chiave. Fornisce un modo per gestire con grazia situazioni inaspettate come questa e implementare la pulizia (ad esempio, smaltire gli oggetti non più necessari e resettare i trigger degli script o altri attributi importanti).

Punti da notare:

1. È buona norma avvolgere il codice in blocchi **TRY:CATCH** per il codice che può lanciare exceptions (chiamate .NET Framework). **QuickScript** incorpora funzioni come **StringToIntg()** ecc. che non lanciano exceptions.

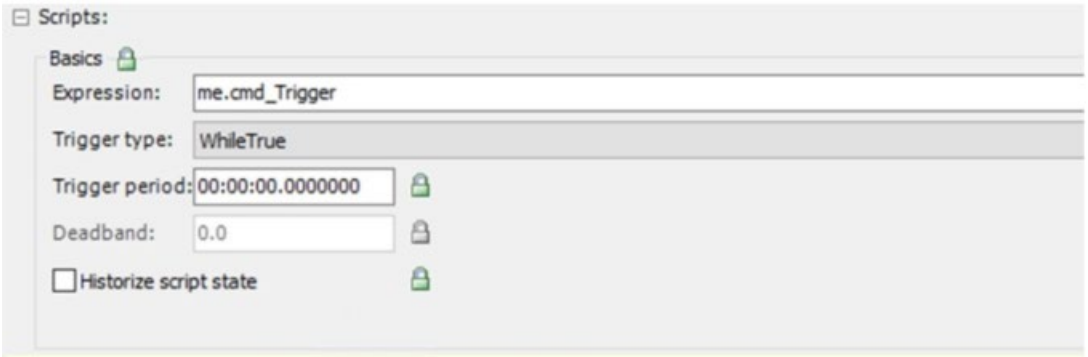
2. È buona norma reimpostare i trigger di script all'inizio di uno script se lo script deve essere eseguito una sola volta. Se lo script deve essere eseguito in loop su periodi di scansione consecutivi, assicurarsi che l'attivazione dello script sia reimpostata nel blocco **CATCH**, in modo che lo script non vada in loop se si verifica un errore.

3. Validare il codice. Verificate sempre la presenza di elementi come i **NULL** quando utilizzate i tipi di **.NET Framework** nel vostro codice. Utilizzate la documentazione online per verificare se un determinato metodo può lanciare un'exception e in quali circostanze.

4. Pulire gli oggetti nel blocco **CATCH**, se necessario. Alcuni oggetti devono essere rilasciati una volta terminati, di solito chiamando il metodo **Dispose()** di quell'oggetto. Poiché l'esecuzione del corpo principale dello script si arresta quando viene lanciata un'exception e salta al blocco **CATCH**, è facile che la pulizia degli oggetti non venga effettuata.

5. Utilizzare più blocchi **TRY:CATCH** per script più complessi. In questo modo si può controllare meglio il flusso dello script e consente una risoluzione dei problemi più intuitiva.

Seguendo i punti precedenti, il codice di esempio può essere riscritto in questo modo:




```
dim handle as System.Data.SqlClient.SqlConnection;

'reset the script trigger
Me.cmd_Trigger = false;

try
    'get the DB connection object from the AppDomain
    handle = System.AppDomain.CurrentDomain.GetData("MyConnection");

    'check if we got a valid object from AppDomain
    if handle <> null then
        'check if the connection is open and able to be used
        if handle.State == System.Data.ConnectionState.Open then

            '.....do some work with the database
            '.....cleanup resources etc
        else
            LogError("Connection state is not open, cannot continue");
        endif;
    else
        LogWarning("Unable to get connection from AppDomain");
    endif;
catch
    if handle <> null then
        handle.Dispose();
    endif;

    LogError("Something went wrong: " + error);
endtry;
```

Riferimenti indiretti

Evitare i binding non necessari negli script. Ad esempio, scorrere un array di stringhe che contengono riferimenti ad attributi locali e poi usare la funzione **BindTo()** per ottenere/impostare il loro valore. Questo è un esempio di cattiva progettazione degli oggetti. È sempre più veloce fare riferimento a un attributo direttamente piuttosto che indirettamente. Sebbene il binding a un attributo locale (**Me.**) sia veloce, ha comunque un tempo di esecuzione più elevato(overhead).

Per contestualizzare questo overhead, si consideri il seguente script che utilizza un singolo riferimento indiretto a un attributo locale:

```
dim sWatch as System.Diagnostics.Stopwatch;
dim pvIndirect as indirect;


Me.cmd_SpeedTestTrigger = false;

try
    sWatch = new System.Diagnostics.Stopwatch();

    sWatch.Start();
    pvIndirect.BindTo("me.PV");

    pvIndirect = 66.3;
    sWatch.Stop();

    Me.TotalTicks = sWatch.ElapsedTicks;


catch
    LogError(error);
endtry;
```

La tabella seguente mostra i tick totali su tre esecuzioni dello script precedente:

Numero di esecuzione	**Ticks Totali**
1	1797
2	1220
3	1310
Totale:	**4327**
Media:	**1442**

Lo script viene modificato in modo da utilizzare un riferimento diretto come questo:

```
dim sWatch as System.Diagnostics.Stopwatch;
dim pvIndirect as indirect;


Me.cmd_SpeedTestTrigger = false;

try
    sWatch = new System.Diagnostics.Stopwatch();

    sWatch.Start();
    Me.PV = 66.3;
    sWatch.Stop();

    Me.TotalTicks = sWatch.ElapsedTicks;


catch
    LogError(error);
endtry;

```

La tabella seguente mostra i Ticks totali su tre esecuzioni dello script di cui sopra:

Numero di esecuzione	**Ticks Totali**
1	82
2	74
3	72
Totale:	**228**
Media:	**76**

Come si può vedere dalle tabelle precedenti, il riferimento diretto è ~19 volte più veloce di quello indiretto. Entrambi gli script sono stati eseguiti in meno di 1ms. Tuttavia, come dimostra questo esempio molto elementare, la referenziazione diretta è migliore di quella indiretta.

Sovraccarico

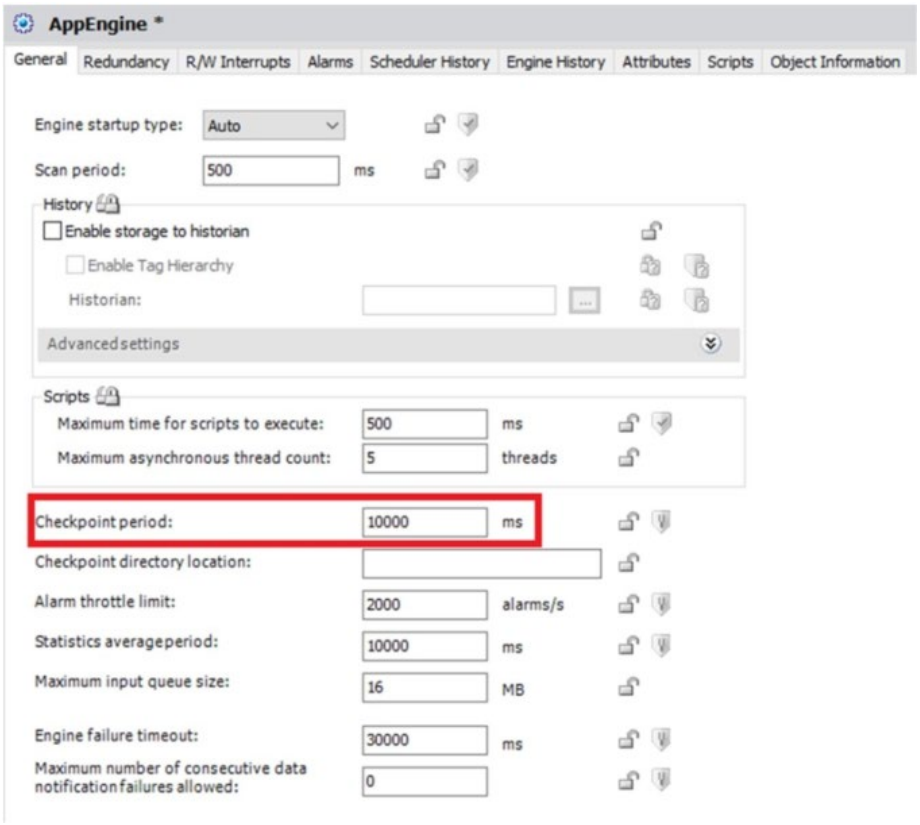
Scan Period & Checkpoint period

Il periodo di scansione predefinito di un Engine è di **500ms**. È necessario valutare se l'Engine ha bisogno di un'esecuzione così veloce. Ciò dipende dal numero di oggetti che il motore ospita e da ciò che gli oggetti fanno durante l'esecuzione. Non esiste una regola ferrea sul numero di oggetti che un Engine può ospitare e si dovrebbe testare spesso l'esecuzione. Dando al motore più tempo per l'esecuzione, si può potenzialmente evitare che si verifichino degli overrun. **Questa semplice modifica della configurazione viene spesso trascurata.**

È facile pensare che, se un'applicazione funziona lentamente, accelerando il tempo di ciclo del motore si risolva il problema. Tuttavia, questo non è il caso, perché si sta essenzialmente fornendo meno tempo al motore per eseguire tutto il carico di lavoro prima del ciclo di scansione successivo. Se l'Engine fatica a eseguire il carico quando funziona a 500 ms, la modifica del periodo di scansione a 200 ms non farà altro che aggravare il problema.

La modifica del periodo di scansione viene di solito effettuata quando le comunicazioni con i dispositivi di campo sono lente. In questo caso, invece di accelerare il periodo di scansione, si consiglia di regolare gli **interrupt di lettura/scrittura** per migliorare la velocità di comunicazione con i dispositivi di campo. Gli interrupt di lettura/scrittura si verificano in punti del ciclo di scansione e possono aumentare la velocità complessiva delle comunicazioni. Inoltre, i periodi di scansione dell'Engine devono essere impostati su **numeri primi**. I numeri primi devono essere diversi per gli Engine in esecuzione sullo stesso host. Questa pratica impedisce ai motori di iniziare l'esecuzione contemporaneamente.

Allo Scan period si affianca il **checkpoint period**, impostato nella finestra **General** della configurazione dell'**Engine**:



Il checkpointing in Application Server è il processo di salvataggio dei valori degli attributi di runtime su disco. Ciò consente la persistenza dei dati non-IO dopo il deploy o failover. Per impostazione predefinita, il checkpoint period è impostato su 10000 ms (10 secondi).

Questo parametro non deve essere 0, perché significa che il checkpoint verrà eseguito a ogni scansione. Questo non è consigliato perché aumenta il carico di lavoro dell'Engine.

È necessario controllare questo parametro perché il valore predefinito potrebbe essere troppo basso per gli Engine sottoposti a un carico pesante.

Frequenza di esecuzione degli script

La frequenza di esecuzione degli script è facile da trascurare, ma è una parte molto importante della gestione del carico di un Engine.

L'esecuzione di uno script, per quanto benevolo, ha un costo, un overhead, in termini di risorse (tempo, CPU, RAM, ecc.).

Quando si parla di frequenza, si parla di secondi, non di millisecondi. Uno script abbastanza innocuo che viene eseguito ogni cinque secondi può, con un certo volume, causare problemi all'Engine.

Considerate quanto segue:

1. Lo script deve essere eseguito periodicamente? Può invece essere basato su eventi?

2. Se deve essere periodico, deve essere eseguito ogni N secondi. È possibile aumentare questa frequenza?

3. L'intero script deve essere eseguito ogni volta? È possibile eseguire solo alcune parti dello script ogni volta? Riducendo il tempo di esecuzione dello script si riduce il carico sull'Engine.

4. Considerare dove viene eseguito lo script. Se questo script fa parte di un template e quindi, per estensione, viene eseguito in tutte le istanze derivate, è necessario? È invece possibile eseguirlo in un sottoinsieme di oggetti?

Ambiente

L'assistenza tecnica AVEVA consiglia di dimensionare correttamente gli **Application Object Server (AOS).** Per le raccomandazioni sul dimensionamento, consultare la **System Platform Installation Guide**.

Quando si dimensionano i sistemi, si consiglia di aggiungere il **25%** come overhead per il **sistema operativo (OS).** La mancanza di risorse sufficienti per un sistema può anche causare l'overrun degli Engine. Inoltre, ogni Engine di un sistema viene eseguito su un singolo core. Il rapporto massimo consigliato è di **2:1 tra Engine e core** (se l'**Hyperthreading** non è abilitato, altrimenti è 1:1), anche se ciò dipende dal carico di lavoro di ciascun Engine. I test sono fondamentali in questo caso.

Inoltre, assicurarsi che i software di terze parti non abbiano un impatto sul sistema. Tutte le **esclusioni antivirus** devono essere applicate vedi guida 'AntiVirus Exclusions for System Platform 2017' e 'AVEVA System Platform 2020 (formerly Wonderware) AntiVirus exclusions'.

Nota: se gli Engine sono stati configurati per avere percorsi (path) **Store/forward** e/o **checkpoint** non predefiniti, anche questi devono essere inclusi nell'elenco di esclusione.

Inoltre, non è consigliabile eseguire il backup di un nodo **AOS**, perché questo può avere un impatto sul sistema. Un backup utilizza in genere la tecnologia **Volume Shadow copy Service (VSS).** Se il software non è **VSS-aware**, come nel caso di Application Server, può causare

instabilità e sovraccarichi, a seconda del tempo di esecuzione del backup VSS. Quanto più velocemente viene completato, tanto meglio è, quindi, un hardware più veloce in questo scenario ma non ci sono garanzie.

Inoltre, per un nodo **AOS**, i dati non sono memorizzati qui e quindi non si ottiene molto con il backup di un nodo **AOS**. Se si dovesse ripristinare il backup, è molto probabile che si debba eseguire una nuovo deploy per far funzionare di nuovo il nodo, vanificando così l'utilità del backup.

Virtualizzazione

Non è raro vedere sistemi sovraccaricati quando si utilizza un hypervisor (VMWare, HyperV ecc.). La configurazione di un cluster virtuale è tipicamente al di fuori delle competenze dei team OT e di solito viene lasciata ai reparti IT.

È molto facile cadere nella trappola di:

"Oh, la macchina funziona male, aggiungerò altre vCPU (core) al sistema".

Questa mentalità deriva da due aspetti:

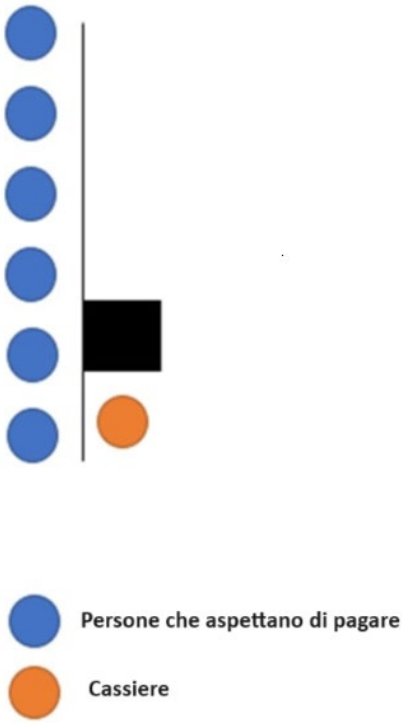
1. Pensiero ereditario: quando le applicazioni venivano eseguite su macchine fisiche dedicate.

2. La facilità di modificare la configurazione di una macchina nell'hypervisor: è molto semplice aggiungere **vCPU**, **RAM** ecc.

Non tutto è come sembra: una **CPU virtuale (vCPU)** non è un core, è un thread e non corrisponde direttamente a una **CPU fisica (pCPU)**. L'aggiunta di altre vCPU a una **macchina virtuale (VM)** non necessariamente migliorerà il funzionamento della VM, ma può peggiorarlo.

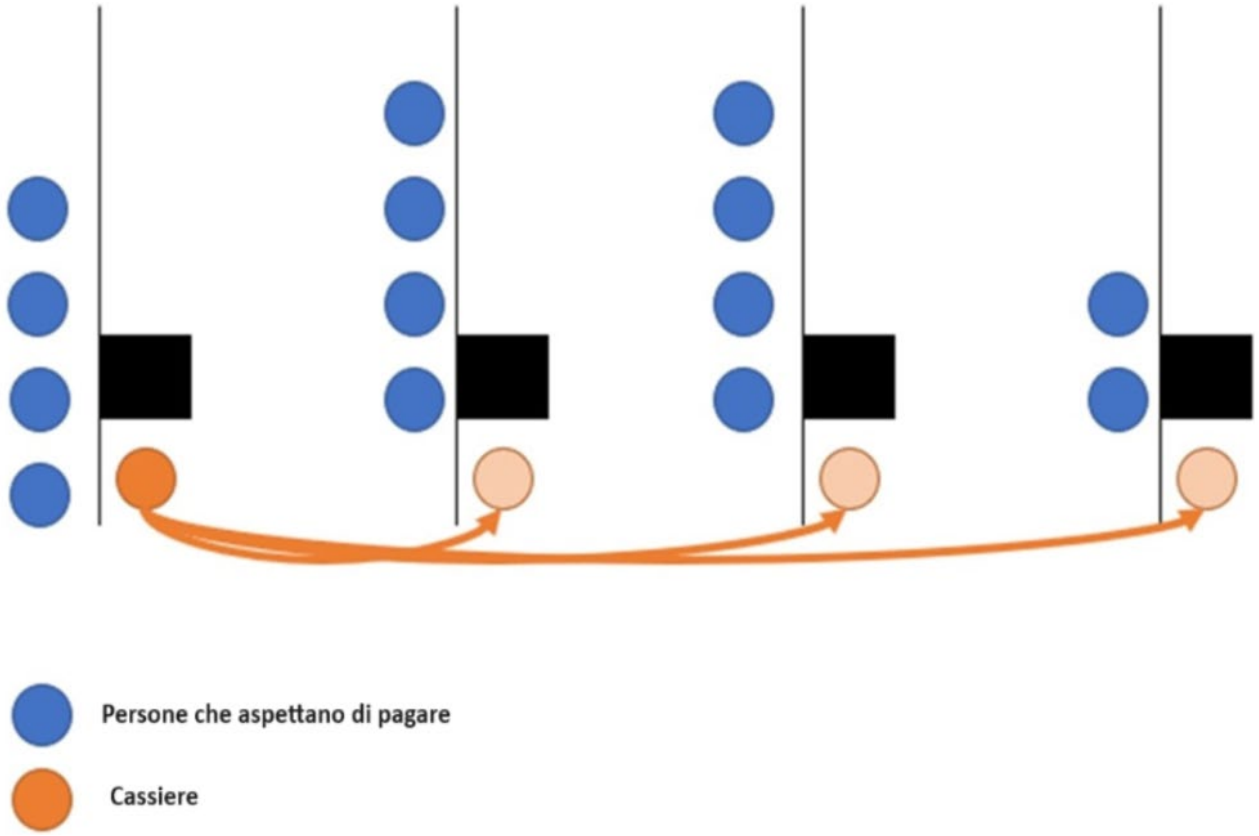
Le vCPU sono una rappresentazione software di un thread (un percorso) che può essere utilizzato per eseguire comandi sulla CPU. Non è una rappresentazione di un core fisico della CPU. Questo aspetto è ulteriormente amplificato dall'**Hyper-Threading**. La tecnologia Hyper-Threading può essere utilizzata per estendere il numero di core, ma ha un impatto sulle prestazioni. Una CPU a 8 core avrà prestazioni migliori di una CPU a 4 core con Hyper-Threading.

Considerate quanto segue:



In questa analogia, consideriamo un supermercato in cui le persone aspettano di pagare alla cassa. Nel primo scenario, solo un cassiere è disponibile per gestire le persone in attesa di pagare. Qui vediamo tutte le persone in attesa di essere servite.

Cosa possiamo fare per accelerare i tempi? Aggiungere altri banchi di controllo!



Sono state aperte altre casse e ora ne sono disponibili quattro. Tuttavia, non abbiamo aumentato il numero di cassieri disponibili, per cui ora il cassiere deve correre tra le casse. Questo aumenta il tempo di attesa delle persone in coda.

- Casse = thread CPU (vCPU)
- Cassiere = core della CPU (pCPU)
- Persone in attesa di pagare = istruzioni da eseguire sulla CPU

In parole povere, non ha senso aumentare la quantità di vCPU in una macchina virtuale se non c'è nulla per alimentarle sul lato fisico; si aggiungerà solo un aumento dei tempi di attesa in quanto l'hypervisor deve destreggiarsi tra il carico di lavoro delle varie vCPU e le pCPU.

Per le macchine process-critical, il rapporto **vCPU:pCPU** dovrebbe essere **1:1.25** o **1:1** al massimo. Questo rapporto può essere impostato nell'hypervisor staticamente nella configurazione della macchina virtuale o dinamicamente tramite **DRS**:

https://www.vmware.com/uk/products/vsphere/drs-dpm.html

Monitoraggio dell'Engine

Il seguente elenco di attributi può essere monitorato in **Object Viewer** o **Historized** per capire come si comportano i singoli Application Engine:

- Engine.AsyncScriptsWaitingCnt
- Engine.AsyncScriptThreadMax
- Engine.Folding.Condition
- Scheduler.ExecutionTimeAvg
- Scheduler.ScanCyclesCnt
- Scheduler.ScanOverrunsCnt
- Scheduler.ScanOverrunsConsecCnt
- Scheduler.TimeIdleAvg

Referenze

- Why Engines Overrun