Scalability Architecture For Processing
using Microsoft .Net Remoting

Seth Nielsen and Dr. Yuke Wang
School of Computer Science, University of Texas at Dallas, Richardson, Texas, U.S.A.

Abstract - Microsoft .Net Framework provides intrinsic capabilities for distributed computing but does not provide prioritization and scaling services. As more applications and database systems convert from mainframe systems to Microsoft .Net Framework, a framework to provide these services is needed. Microsoft provides for some of these services with more advanced offerings, but with increased operational requirements and licensing costs.

This paper will define the architecture for a software-based solution that will require only .Net Framework and SQL Server to provide scaling, scheduling, prioritization, logging and load balancing services. This solution provides for horizontal scaling of defined processes that can further break themselves into sub-processes and submit these to be run on additional physical servers without the calling application being aware. The load balancing solution described here will not only attempt to locate the best fit server for the requested process, it will allow for processes to be assigned only to servers defined as capable to run the process.

Keywords: Software Architecture, Web-based Applications, Distributed and parallel systems

1 Introduction

Microsoft .Net Framework is a common software solution and provides many higher level functions. However, these solutions do not naturally provide many of the services available to mainframe systems. This includes scaling, scheduling, prioritization, logging and load balancing services. Microsoft and other vendors provide large-cost solutions, but these require large expenditures of hardware and pre-built software solutions that have large annual fees and maintenance fees. As companies move away from mainframe systems where process scaling is handled intrinsically, modern software solutions need to provide these same services.

1.1 Objectives

This paper will provide a mission-critical process scaling architecture that utilizes only Microsoft .Net Framework and Microsoft clustered SQL Server. This will allow more software systems to provide scaling and load-balancing services for CPU, memory and query intensive processes with quick turn-around for fast and short processes at the same time.

1.2 Why is this important?

Many smaller to mid-size companies have found that the mainframe implementations are too expensive to purchase and maintain. Microsoft has provided a relatively inexpensive software solution that does not provide standard mission-critical solutions for their growing needs. This paper will describe a solution that can be implemented by the typical software development team utilizing the tools they are most familiar – Microsoft .Net Framework and Microsoft SQL Server.

1.3 Brief Summary of Existing Approaches

There are several third party solutions that currently provide scalability services, but they are either high priced or do not meet the mission critical needs of high availability.

Microsoft provides several solutions in this regard. Network Load Balancing (NLB) Services are offered as an extension of Windows Server 2008 [14]. NLB offers fault tolerance and will allow scaling to 32 servers per segment, but does not ensure that the destination server is running the service needed [16]. With no load balancing mechanism built in, the solution does not offer enough benefit for the software team not already using NLB for other purposes.

Microsoft also offers SQL Service Broker (SSB) as part of Microsoft SQL server tools. SSB provides a messaging and queuing solution that uses the same transactions utilized in data modifications to provide distributed processing [11]. However, the message queue allows communication in a first-in, first-out order so does not easily provide for the prioritization services that are needed for a scalable solution. The SSB solution also requires most of the communication logic to be stored in stored procedures on the SQL Server, which makes debugging more difficult.

Microsoft MQ (MSMQ) [17] and IBM WebSphere MQ [12] are also solutions for scalability. The intent of message queuing systems is to break down processing into very small pieces and handle each item individually within the queue. This requires a significant amount of work to then collate all
returned information together and does not work well when significant amount of preparation/data access is needed to complete the processing. Both MSMQ and WebSphere MQ are expensive solutions which require a significant amount of management and oversight [17].

1.4 Summary of the Remaining Paper

The remainder of the paper will include the following sections. Section 2 briefly describes the Microsoft solutions to be utilized by this solution and an overview of scalability. Section 3 describes the Process Scaling Architecture solution and shows how it meets the needs as discussed here. Section 4 will show the results and Section 5 provides the conclusions.

2 Background

2.1 Microsoft .Net Framework

Microsoft .Net Framework developed by Microsoft Corporation provides a consistent object-oriented programming environment using a common language runtime (CLR) and automatic garbage collection [18]. The framework provides a substantial amount of tools for development teams.

2.2 Clustered Microsoft SQL Server

Microsoft SQL Server 2008 provides data storage and retrieval for mission-critical applications that require high levels of availability and performance. Failover clustering provides protection against planned and unplanned downtime [7]. Microsoft SQL Server clustering provides for two separate physical machines to be utilized for handling all incoming requests. If one server fails, all of its services are transferred to another machine without the calling applications being aware of the transfer [15].

2.3 Microsoft .Net Remoting

Microsoft .Net Remoting provides an abstraction layer to the developer so processes can be run on remote servers without significant involvement in this communication [8]. The .Net Remoting service provides the serialization (marshaling) of data prior to sending the request to the remote machine and then performs the deserialization at the remote server before the remote process begins its processing. The remote process will run solely at the remote server performing the logic of the remote object and utilizing the resources of the remote server.

2.4 Scalability

Scalability is the ability of a software application or software systems to be able to perform in expected ways with reasonable performance as the demands of the software and systems are increased. There are two main approaches to allow for applications and systems to scale.

Vertical scaling (or scaling up) requires the hardware components of the server(s) to be upgraded or improved. This can involve upgrades to processors, memory, disks, and network adapters [10] to reduce bottlenecks to the overall application as demands increase. Vertical scaling works best when the application is limited to run on the currently defined servers.

Horizontal scaling (or scaling out) requires additional servers to be added to the environment. This spreads the application and its processing across more resources. These improvements allow for growth to happen gradually and utilize less expensive servers [10]. Applications that can handle horizontal scaling also allow for these servers to be taken off-line during maintenance without impact to the system and help provide for mission-critical demands to keep the applications running during hardware failure.

3 Possible Solution

This paper is focused on a design to provide a cost effective solution to the many needs of mission-critical process intensive applications. These applications will generally be required to fulfill requests for the following conflicting needs:

1) Long running processes with significant data access that will utilize the available resources
2) Short running processes to fulfill a single request for the user that expects results immediately

3.1 Indiscriminate Scaling Solution

One way to fulfill these differing requests is to make them both function in the same way. Adjusting long running processes into many short running processes could make them both the same, but this would cause some negative side effects.

Indiscriminately forcing all processing to be short running processes will significantly increase the overhead of long-running processes that used to share the preparation work of accessing SQL data optimized for long running processes. Running all work as short running processes could significantly increase network bandwidth and increase context switching on each server. In addition, the long running processes need to provide a single aggregated response to the application. This proposed solution would not provide for this collation of data.

3.2 Process Scaling Architecture Solution

A better solution is to create a structure that can run both the long running processes and the short running processes in a way that best fits their needs. The short running process needs to be given priority over the long running process and available resources must be reserved for these requests. The
long running process needs to be able to intelligently break itself into smaller sub-sets in a structure that can provide a single logging mechanism and a pre-defined structure for collating results back to the main process.

The Process Scalability Architecture (PSA) solution provides this staged scaling solution that allows the long running processes and short running processes to both run to their differing needs. The main calling application will simply request for the process needed to be run and PSA will submit the process to be run on the best fit server and with the scheduling and priority as requested.

The staged scaling occurs within the long running processes as it can utilize the same process request structure as utilized by the main application to request sub-process to be run for the main process. These sub-processes, called worker threads, are pre-defined to be grouped together with the main process in terms of logging and result storage.

This grouping allows the worker threads to store their results in the same structure as the main process allowing the main process to collate this information and provide a single response back to the calling application. This grouping also allows for the main process and each worker thread to report any errors, warnings or informational data into a single logging location for the end user.

Figure 1. Process Queue Log

The logging mechanism, as is shown in Figure 1, will provide for all messages related to the main process to be displayed together (using the same Root Process Queue ID) but still allow the end user to see the details of which worker thread performed the process (based on Process Queue ID).

The Process Scalability Architecture is a complete solution that provides all of the services needed by mission-critical applications:

1) Horizontal scaling of processing
2) Load balancing to best fit servers
3) Higher prioritization of immediate requests
4) Scheduling to run processes at non-peak times
5) Consistent logging for viewing and monitoring

3.3 PSA Configuration

As is shown in Figure 2, the solution involves three different physical groups of servers. The process server farm includes servers able to perform the processing needs of the application. The clustered SQL Server provides the data storage for process information and process server status. The Client is any software application utilizing the Process Request Agent to run a process. This could be a web farm, stand-alone application and the process servers themselves.

3.4 Process Server Service

The Process Server Service will be deployed to each Process Server and will be responsible for listening for requests to run defined processes. The Process Server Service will be responsible for knowing which processes are allowed to run on the server and any restrictions on how many processes of each type can be run.

3.5 Process Manager Service

The Process Manager Service will be deployed to a subset of the available Process Servers and will be responsible for locating the best Process Server to utilize for any pending requests. The Process Manager will poll the SQL Server for both requests that are scheduled to run immediately and requests that are deferred.

3.6 Request Listener Service

The Request Listener Service will be deployed to all servers and will be responsible for communicating between the Process Manager and the application that has requested to run a process. This service is critical as it gives the Process Manager a single access point to each server even if the server is running multiple applications.
3.7 Process Flow Overview

The Process Scalability Architecture, as shown in Figure 3, is separated into four logical groups: The Process Server, the SQL Server, the Load Balancer (Process Manager) and the Client. It is not represented in this diagram, but the Process Server can also play the role of the client — this is how it breaks its running process into smaller requests and scales to other process servers.

The diagram above is utilizing arrows to show all paths of communications between the entities. This communication is performed using .Net Remoting (for any communication between services) or transactions on the SQL Server (for all communication with the SQL Server).

This process flow will show the initialization of all servers, the initial request to run a process, the determination of where to run the process, and the completion of the process.

3.8 Initialization

As each service is started, it is responsible for registering itself with the SQL Server. During this registration process, it will record information about itself on the SQL server and collect information on all other servers available in the environment. This helps the service know which servers to attempt communication with, but the process flow will also handle the scenarios when new servers are added or when existing servers fail or are taken off-line.

3.9 Process Request Agent

The Process Request Agent (PRA) is utilized by any client application to request a process to be run. The PRA is a middleware object that performs the necessary .Net Remoting calls to request a reservation on a process server and then runs the process for the calling application. All .Net Remoting calls are abstracted by using the Broker design pattern [6] that creates a client proxy and a server proxy for the methods utilized by the development team.

The Process Request Agent provides a single interface to allow the application to request a process to be run, define when it should be run and the parameters needed to run the process. With this information, the PRA will determine the priority of the process and submit this request to be run by recording the request in the SQL Server. If the process is scheduled to be run at a later time, then the PRA will let the application know that the request has been submitted. If the process is to be run immediately, the PRA must then remote to the Request Listener Service to await the determination on which process server to run the process.

3.10 Process Manager

The Process Manager (PM) manages both the queue of requested processes and the status and load of each process server. This queue is stored within the ProcessQueue table as shown in Figure 4. In combination with ProcessQueueStatus, it determines the available processes and their scheduled start time. The PM will periodically query these tables and sort the processes based on Priority and the original submission time.
Once the Process Manager locates a process to be run, it will attempt to find the best-fit server to run the process. To do this, the PM must know which servers are online and available, which servers are able to run the process, the available resources on each server and what processes are currently running on each server.

These needs are handled by creating an open channel of communication between the Process Managers and Process Servers. The Process Server uses this channel of communication to update all Process Managers with its status and processing load when it starts, each time it picks up a new process and periodically while running each process. If the Process Managers or Process Servers cannot communicate with each other due to network issues or otherwise, the communication failure is logged but does not impact the processing. The Process Server status and load information is also stored in SQL tables, as is shown above, so the Process Managers can reset their status periodically or during initialization.

Of the available process servers that can run the process requested, the Process Manager will find the Process Server with the least activity and most available resources. This determination can be made by comparing the available resources of the server against the utilized resources. This paper will not detail the load balancing mechanism utilized, but the data required to make this determination is managed within the current solution.

Once the Process Manager determines the best fit server, it will request a reservation from the Process Server. This reservation allows the Process Server to have the final determination of whether to run the process and allows time for the Process Manager to communicate back to the calling application. Once a reservation is returned from a Process Server, the Process Manager communicates this back to the Request Listener Service which helps the Process Request Agent to run the requested process on the best fit Process Server.

### 3.11 Process Server

The Process Server’s (PS) main role is to run the requested process. Along with that role is the responsibility to ensure that only the allowed processes are run on the server and within the allowed limits defined for the server. The Process Server makes this determination based on its configuration that is stored within SQL tables, as is shown in Figure 5.

The Process Server can be constrained to only pick up a certain number of each process type (defined by ProcessServerCapability) or can be defined to not accept any new processes once available resource levels are low. It can also reserve resources and available process capability for the short running processes so these can run at any time. As the Process Manager requests a reservation, the Process Server can still decline and the Process Manager will look to the next best server to run the process. If the reservation is granted, then the Process Server will reserve the resources or processing limits for a limited time before the reservation expires.

For each process run by the Process Server, the process can determine if and when it needs to allocate some of its work to other worker threads. Once determined, the sub-elements of the process are requested to be run by another Process Server in the same way that the main calling application requested a process to run – repeating the same process again. The only difference is that these worker threads will be linked to the
root process so all logging and activity is linked to the initial request.

While the Process Server is running the process from the calling application (or from another process), it will store all of its running information into the shared ProcessQueue, ProcessQueueStatus and ProcessQueueLog tables as discussed above. This unified storage location allows all logging information to be viewed together for the root Process requested by the end user. This allows the user to view each process’s status and performance across all available servers.

3.12 High Availability

As the PSA is intended for mission-critical applications, it provides high availability. First, all elements of the architecture besides the calling application are built using Windows Services. This allows the process itself to run faster as it does not need to handle Windows messaging that occurs for applications. It also allows the operational teams to configure the services to automatically start when the machine turns on and automatically restart any time the service fails due to SQL failures, etc.

Second, each of the Process Servers and Process Managers are built to handle the addition and deletion of servers without any impact to the end users’ request to run a process. Each server will broadcast their status to all other servers in the environment each time they start or stop offering services. Additionally, the Process Manager attempts to maintain the status of all Process Servers in memory for faster service, but will periodically rebuild its in-memory data from the SQL.

Third, there is no single point of failure as the structure allows for multiple Process Servers and multiple Process Managers to be running at the same time. This allows the operational team to add and remove servers from deployment as needed. For any mission-critical application using SQL Server, the operational team will already have handled deploying the SQL Server as a clustered service with a possible second hot data center that will automatically become active on failure.

4 Validation

The implementation of this solution has shown very positive results. The end product provides a high available solution that can run short processes quickly, run long processes efficiently, and provide a standard mechanism to scale these long processes without any significant impact to the end user.

4.1 Case Example

Prior to this implementation, users were struggling with a process that would take 24 hours to run to completion (assuming no failure occurred). This timing was not sufficient for their needs, so they would need to manually split out the processing into multiple submissions of lesser size. This would allow the process to complete in six hours but require an additional two hours of manual effort to split up the processing. In addition to the time required, the user could create issues by not splitting up the process correctly.

With this improvement, the user can submit the full process and the built-in scalability allows all of the initial data work to occur in the main process and allocation of 12 different worker threads to complete the process. This allows the whole process to complete in a little over 3 hours with no extra involvement of the end user. Even if there is a problem with one of the worker threads (by user error or system error), the user can then rerun a single worker thread and the logging of the rerun is included with the original process submission.

4.2 Validation By Bottlenecks

Another way to validate the success of the implementation is by reviewing the processor utilization on the Process Server before and after the implementation of this improvement. Prior to this implementation, the long running process and the manually created smaller processes would spend a significant amount of time waiting for responses from SQL requests. The queries were efficient with good execution plans, but there were a significant amount of requests. Due to this waiting time, the overall processor utilization would rarely go above 20% utilization. With these improvements, the utilization now peaks to 80%.

In addition, the SQL Server interaction increased so much with this improvement that the query and connection time out settings for the Process Server and the SQL Server had to be adjusted as the application was now making SQL Server work harder. This shows that the application and the software are no longer the bottleneck and operational improvements to hardware can now make improvements to the overall performance of the system.

4.3 Other Issues Identified

There are other possible issues in this implementation that should be reviewed when utilizing this solution.

First, remember that a 32-bit installation is still restricted to 1.5G for a single Process Server service. This means that deploying to 64-bit servers is the preferred deployment.

Second, this solution does not automatically protect against deadlocks from occurring. The Process Server Capability configuration must ensure that there are as many worker threads available as the number of main threads available. This will ensure that each main process will always have at least one worker thread to perform its work and eventually complete.
Third, starvation protection has not been built into this solution. This was not a significant enough issue for the initial implementation, but this can be added later by having the Process Manager increase the priority of all pending Process Queue records each time it does not have an available Process Server to run it.

5 Conclusions
This paper has defined a Process Scalability Architecture to be used in mission-critical applications with processor-intensive needs. This solution provides efficient solutions for common needs of these applications including: horizontal scaling, high availability, load balancing, scheduling, prioritization, consistent interfaces, and unified logging of all activity. The horizontal scaling mechanism also provides the ability for each process to also break out its processing to sub-processes on different servers. This solution has been implemented and validated against a mission-critical application and the identified issues have been corrected.

6 Acknowledgement
The authors would like to thank Craig Burma for his help and support during this work. They are also very appreciative of Ben Horstman, Daniel Wells, Mark Lansford and all of Seth’s co-workers who helped to work through the issues related to this solution and to validate the design.

7 References