22nd International Congress on Modelling and Simulation, Hobart, Tasmania, Australia, 3 to 8 December 2017
mssanz.org.au/modsim2017

Methods of Distributed Processing for Combat Simulation Data Generation

Lance Holden[a], **Scott Russack**[a], **Mohamed Abdelrazek**[b], **Rajesh Vasa**[b], **Shannon Pace**[b], **Kon Mouzakis**[b], **Rhys Adams**[b]

[a] *Defence Science and Technology Group*

[b] *Deakin University*

Email: lance.holden@dst.defence.gov.au

Abstract: Combat simulation requires an extensive amount of data to be generated for the execution of the simulations followed by an extensive amount of iterations of the simulation tool to produce quantities of data sufficient for analysis. Typically this generation process exceeds the capabilities of a single desktop computer to perform in a usable time period. Effective data generation requires a method of harnessing the power of multiple computers working towards the same goal.

To meet the data generation requirements of combat simulation execution a series of distributed processing architectures were developed. These have expanded from specific task solutions to generic task distributed processing architectures. Each implementation has to solve the issue of distributing processing tasks that were not initially developed with an existing distributed processing framework in mind (such as Map-Reduce). Each of the architectures has been built on experiences learned from previous implementations. These lessons have resulted in two architectures available for our distributed processing needs.

These architectures take a different approach to the distribution of jobs and the management of work execution and scheduling. The first is a Distributed Queue architecture that is based on a dynamic client pool of processing nodes that pull jobs from a well-known job description queue using transient data transfer. This approach allows cross-platform processing nodes to be added to the distribution network on a needs basis. The other is a Distributed Scheduler architecture uses a resource scheduling algorithm to distribute jobs to resources on a network. This scheduler can manage task dependencies and the transfer of persisted data between processing tasks. Both implementations have to manage for resource nodes not working correctly, processing errors on the remote tasks and monitoring the progress of any assigned tasks.

This paper will examine the history of the distributed processing architectures we have used. It will describe the two resulting architectures and the differences between them. It will then outline our selection criteria and the currently used distribution implementation and future improvements to the process.

Keywords: *Distributed processing, data generation, combat simulation*

1. INTRODUCTION

Land Combat Analysis (LCA) branch at Defence Science Technology Group (DST) requires a large amount of generated data to provide relevant information for studies. Data is generated to provide entity interaction for combat simulations and data is produced by the execution of those combat simulations. The computer processing power required to generate the data exceeds the capacity of a single computer within time frames suitable for report delivery. DST has had several attempts at distributing work across multiple computers to provide extra processing capability. Each required developing of bespoke solutions, but these were only useful for a specific data generation tool.

Recognising that there were many similarities in the requirements and implementations of each bespoke solution DST started to look for a solution that could easily be adapted to any future task that would benefit from distributed processing. DST defined a set of requirements and a partnership was established with Deakin University to develop an implementation of a distributed processing framework. The framework needs to manage job scheduling, the transfer of data and be able to operate in a dynamic environment where processing resources may have restrictions in when they could be accessed. The framework also has to support running existing tools not designed for distributed processing on a variety of operating systems.

Due to the need to support immediate data generation needs two implementations were started, one based on existing work at DST called the Distributed Queue and another in partnership with Deakin University for the Distributed Scheduler (called Virgil). Each implementation had a different approach to managing the distribution of tasks across processing resources.

As both implementations reached a point where they were suitable for use in distributed processing DST had to decide which system to support. A set of selection criteria was required to determine which would be the best to support the data distribution. This paper outlines how each implementation was ranked and which solution was adopted for use and where the future development effort will focus.

2. BACKGROUND

DST has had several different solutions developed for distributed processing depending on the simulation tool being used.

2.1. CAEnQueue

The Close Action Environment (CAEn), [Shine et al. (2007)], is a stochastic simulation that can be run in a Monte Carlo simulation mode. It can use and modify a configuration file for batch processing but, using this method of running jobs on multiple computers with the same file, can cause concurrency issues resulting in jobs being skipped. As well as missed jobs, stability issues with CAEn caused it to intermittently fail. As a result DST produced a system called CAEnQueue to manage the concurrency issues and provide error recovery while running CAEn jobs in a distributed environment.

The CAEnQueue system uses Java Remote Method Invocation (RMI), described by Oracle (2017), from the Java language for communication between the server managing the jobs and the clients that run the simulation. The server was a single Java application with a web front end to allow users to manage the jobs. If the server was restarted all the clients would also need restarting and jobs submitted again. This implementation required manual configuration of all the machines used in the processing network.

As CAEnQueue continued to be used to run all of the simulations it was improved to include management of individual jobs and sets of jobs; improved error handling and job resubmission and priority management.

2.2. Simulation Queue (SimQ)

CAEnQueue was written specifically for running CAEn so when DST started to use Combined Arms Analysis System Tool (CXXI) a new system was needed. The new solution was planned to be simulation agnostic so it could support multiple tools. It was to retain the web interface for control but provide more persistence for work that had already been submitted.

The framework to replace CAEnQueue was called SimQ (Simulation Queue). SimQ is a Java application that uses the Java Message Service (JMS), described by Oracle (2013), to connect the client and servers. This library was selected so that a custom message passing framework didn't need to be developed. The JMS requires software called a broker to manage the messages. The ActiveMQ broker from the Apache Software Foundation was selected because it was available as Open Source and had good online documentation.

The implementation used a mix of communication methods between clients and the server. Clients would receive jobs via JMS messages but would receive and transmit the data of each job via a separate TCP/IP stream. The server would talk directly to clients by communication queues but required broadcast topics to coordinate what clients were available. The mix of different communication methods made SimQ fragile if network difficulties occurred.

As with CAEnQueue, SimQ had the ability to rerun jobs that failed. It recorded all active jobs in a database so that the system would return to the last state if the server was restarted. The implementation of this feature wasn't stable at recovering connections to working clients during production level runs. If the server stopped, all clients and the active job had to be restarted. This unreliability had major impacts on the ability to generate data in the required timeframe.

Although built to be a generic distribution queue SimQ only ever supported a CXXI client.

2.3. Simulation Repository (SimR)

To improve the management of the data that was generated for combat simulations the Simulation Repository (SimR) software, as described by Holden (2016), was developed. This tool uses algorithms to generate interaction data between entities in the simulation. The amount of interaction data increases with the number of entities included in the simulation.

SimR required a similar distribution method as SimQ but needed more flexibility in the control message payload and didn't require any message persistence management. Multiple distribution frameworks were built based on the JMS design used by SimQ. Each implementation was similar but bespoke, changes in one version did not automatically flow through to the other implementations resulting in a lot of time debugging and fixing the same issue.

DST began using a virtualized environment via VMware and a Blade server at this time to provide more processing power. Some of the data generation tasks could be used on Linux virtual machines hosted by the Blade but other tasks still required the Windows platform. This was the first experience in using distributed processing on Linux computers and allowed DST to explore the benefits or remote machine configuration and control via virtualization.

3. REQUIREMENTS SPECIFICATION

These implementations of distributed processing gave DST an understanding of what was needed to fully utilize their networked computing environment.

All initial solutions were using the Windows operating system as the main computing platform for the distributed processing. Restrictions in the operating environment meant services couldn't be used to automate the configuration and starting of the worker client applications. The solution to manually start the processing environment required a user to physically access each computer. This would require a single user to manage a large pool of machines or training a large group of users on how to connect their desktop computer to the processing pool. A solution was desired to reduce the amount of configuration or interruption of normal machine use required to add it to the distributed processing pool.

The use of personal desktop computers reduced the timeframe available for most data generation to non-business hours. During this time the processing would be unmonitored and any errors could only be corrected in the following day causing the loss of processing time. A solution was required that would add a pool of computer processing power alongside the normal desktops or replace them entirely. Virtualised computers were determined to be the answer for this. DST purchased Blade servers to provide virtual computers for the processing during business hours and maintain the ability to add idle desktop machines in non-business hours.

The initial SimR implementation's difficulties with managing the software distribution code showed that a unified distributed processing framework was required for ensuring all components benefitted from fixes and improvements to the code base. The SimQ implementation showed that a generic framework was the most viable solution to handling future distribution cases. The generic framework had to focus on message passing only; building in details of what the message payloads consisted of limited the scope of the framework.

Based on these identified needs Deakin University was approached to partner with DST in delivering a system that met the requirements. This was to be the Distribution Scheduler solution that treated a large amount of distributed processing power in a similar way that an operating system schedules processes and threads on its internal hardware. During the time required to develop this system DST continued to require a working system to meet internal demands and so DST continued the design used for SimR, using lessons learnt with Deakin University, into a unified implementation called the Distributed Queue.

4. DISTRIBUTED QUEUE PROCESSING

The DST implementation of the distributed processing system was created by extending the SimQ and SimR solutions. This uses the Java Messaging Service (JMS) to dispatch messages from a well-known queue to any dynamically registered worker client. Workers can request specific types of work and decide how long they continue to take jobs. The task submitter and worker communicate directly to receive the full work specification and to transmit the generated results. Figure 1 provides an overview of the queue distribution work flow.

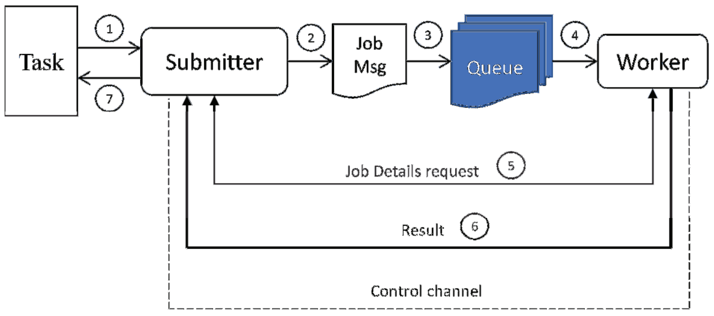

Figure 1. Queue distribution work flow

The work flow starts with a software program requesting a task to be distributed. It will have a single submitter service embedded in the software. The task is passed to the submitter (1) which then converts the job into the message format compatible with the workers (2). A job announcement message is passed to a well-known queue (3). Any number of worker clients will be connected to the queue and one will receive the jobs announcement (4). The worker then communicates directly to the job submitter and retrieves the full parameters for the work (5). After the task has been calculated the results are put into a response and sent back to the submitter (6) which then decodes the result into the data type expected by the task before returning the calculated result (7).

The task is broken into two message parts, announce and parameters. The announce message is small and contains the information needed for a worker to filter jobs. These messages are small because they can be stored in the queue for a long time (the distributed messages are often large in data and duration while JMS is designed to quickly handle small messages). The announce message contains the information needed to let a client know if it can perform a job, for example a worker requiring a Windows application would only accept those jobs if it was running on a Windows computer. The full parameters for a job are transmitted on a private queue established between the worker and submitter when the worker actually starts on the task. Both messages can be transmitted in a mix of text and binary data. The submitter and client need to provide the message encoding and decoding, the Distributed Queue just provides the connection and message transfer.

The submitter provides a level of error recovery by tracking how often it receives messages from the worker. If the worker doesn't communicate within a set time, the job announcement message is resubmitted. The announcement is given a higher priority so that it'll be run as soon as possible, otherwise it would have to wait until all the previous jobs have completed in the queue which could cause a large delay to completing a job.

A worker can check that a submitter still wants a task to be calculated when requesting the details. If the submitter is no longer reachable it is assumed the data generation is no longer wanted. This avoids wasting processing time for unused data. The worker will also wait for a fixed time for the job details; if they aren't received it will discard the job and get another one. This prevents a worker stopping if a submitter isn't sending job details. The job announce message is lost and the task submitter will never complete it, however during design it was decided that submitters will have already failed for this error to occur so no error recovery would be possible.

The basic queuing mechanism doesn't support any load balancing of jobs to workers. Each worker currently accepts one job and performs the work and the user starts the desired number of them per client machine.

Each client machine that runs workers will have to determine the maximum workload it can support and run that many worker instances. An alternative implementation is to have one worker client that accepts jobs until a threshold of current resource use is reached, when it will pause until more local capacity is available. This system would have limited support for ensuring big tasks aren't starved out by lots of little tasks executing.

The DST Distribution Queue does not provide persistence of submitted jobs or results in the task submitter.

5. DISTRIBUTED SCHEDULER PROCESSING

Deakin University approached the distribution problem by having a cluster of known worker nodes and a scheduler responsible for distributing jobs to them. The Distributed Scheduler can handle dynamic workloads and infrastructure changes in real-time. An Open Source implementation of this design was released as Virgil, described in detail by DSTIL (2017). The Virgil system relies on Open Source frameworks to reduce development time and to gain the advantage of having an expanded community of developers.

A Virgil managed computing cluster has two types of nodes: leader (master) nodes, and worker (slave / computing) nodes. A node is a physical or virtual machine running Linux operation system. A leader node is responsible for coordinating work done on the compute nodes. Control of the leader node is managed by a HTTP command server. A slave (compute) node is a node that can execute a job. Compute nodes offer their resources to the leader node(s), and can accept work assigned to them. Figure 2 shows the high-level architecture of Virgil.

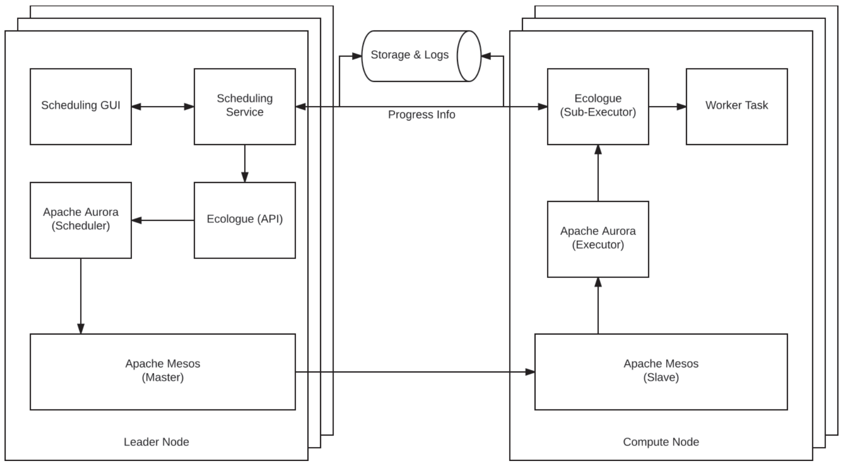

Figure 2. Virgil Architecture and Execution Flow

The Distributed Scheduler is developed on top of the Apache products Mesos, Aurora, and Zookeeper. Mesos is the most widely-used and stable open source cluster management platform. It is resilient to any changes in the underlying infrastructure including adding or removing nodes to and from the cluster in real time. Job scheduling is not a part of Mesos platform, it needs to be managed by another framework, which gives flexibility to implement custom job scheduling algorithms in the future. Mesos provides offers of resources available on the compute nodes to a resource scheduler framework (application). A scheduler can then decide what resources and which jobs to deploy and inform Mesos where to initiate the appropriate jobs.

There are two key open source Mesos Job Schedulers: Marathon and Aurora. They both have similar features. Marathon is easy to use and install, which is not the case for Aurora. Aurora provides a clear job lifecycle (deploy, run, and cleanup events), and additional reliability and bookkeeping features in case of job failure. Deakin decided to leverage useful aspects of Aurora as a job scheduling framework and developed Eclogues to provide some missing features such as inter-job dependencies. Eclogues has server and sub-executor components; the former provides an programming interface for the leader nodes, and the latter is called by the Aurora executor to execute and track the job executable (e.g. the worker task).

Virgil uses Apache Zookeeper as a distributed coordination service. Zookeeper provides necessary primitives to run distributed application including synchronization between cluster nodes, configuration maintenance, groups and naming. Zookeeper instances form a quorum which allows for the loss of a minority of instances while maintaining availability.

There are several improvements still identified for Virgil development.

Currently Virgil requires executables to be manually deployed on worker nodes. A future goal of development is to package and automatically deploy programs on worker nodes based on specs of a given job. This would reduce the possibility of errors due to missing libraries, version incompatibility, etc. Virgil will have an internally-hosted repository of software containers.

The Virgil Job Description Language (VJDL) will be developed to provide a robust job description language to help users specify their jobs, lifecycle, necessary infrastructure, failure handling procedures, etc. This will work with the containers repository and simulation pipeline automation to ensure worker nodes are correctly configured to execute the assigned tasks.

Virgil will be unlikely to be ported to a Windows environment because many of the components are based on Linux built systems. This was done because the target environment for Virgil is a clustered computer and Linux solutions lead in the environment.

6. DISTRIBUTION SYSTEM SELECTION CRITERIA

At the completion of the Deakin collaboration DST group had two distributed processing options available, the Distributed Queue and Distributed Scheduler, but would only have the resources to support one. Selection criteria were created to determine which system would be used.

DST set the following requirements for the selected system:

- Minimal configuration of a dynamic distribution environment
- Supporting operating system specific worker clients
- No modification of existing code
- Result recovery/permanence
- Task Scheduling

The strongest requirement was the need to support distributed processing on the Windows operating system to support the tool chain being used by DST. The Distributed Queue was the only framework able to deliver this capability. On all other conditions the Distributed Scheduler implementation would match or improve over the Distributed Queue implementation. Removal of the requirement for Windows execution and a large enough pool of permanent computing power available would make the Distributed Scheduler a preferred solution.

Each of these points is described in detail below:

The Distributed Queue system needed minimal configuration of the distribution environment. The worker clients only required a Java Virtual Machine (JVM) and a copy of the worker client software, allowing Windows and Linux systems to be used as well as normal desktops or virtual machines. The server requires a single JMS Broker. The server doesn't need to know about the clients when it starts. The Distributed Scheduling method requires a pre-configured Linux only environment. Processing nodes can be added to the network only by reconfiguring and restarting the servers. The Distributed Queue system is the most suited to DST network needs.

At the time of the assessment DST required some programs that only ran on Windows to be part of the distribution environment. While future changes to the libraries used by the Distributed Scheduler may allow Windows support, the Distributed Queue system is the only one capable of meeting the requirement to support clients on Windows. It is also likely that future work on the DST software may remove the need to operate on the Windows platform.

The Distributed Queue system provides no load balancing on worker computers and could cause errors by consuming all CPU or memory. The Distributed Scheduler approach avoids this by specifying how many resources a task is to consume but has the problem that is the resources aren't correctly estimated the task will be killed for over-consumption. The scheduler software components also consume CPU as an overhead that isn't used for anything which reduces the amount of processing power available. Neither solution is ideal, but the Queue system was easier to manage by not having to correctly guess the amount of CPU and memory to be used per job.

Both implementations don't require modifications to existing code but do require the development of wrapper functions to invoke existing software. There was no benefit for either option.

The Distributed Queue system provides no permanence of task data, either the parameters or the results. It was designed to only pass the messages. If the job submitter fails then any already completed results are lost and no additional processing will be done. The Distributed Scheduler system stores results for later retrieval with clients having to request data is deleted after it is used. The permanence of results and the scheduling mechanism makes the Distributed Scheduler the best solution for the result recovery criteria.

The Distributed Scheduler system provides a robust job scheduling algorithm and can coordinate dependencies and tasks that produce other tasks. The Distributed Queue system has a simple scheduling mechanism of first in, first out (unless being repeated). Both solutions can spawn new jobs to calculate sub-tasks for a job. The Distributed Scheduler allows for the sub-tasks to be created and then the active job finishes with a new job collecting the sub-tasks results. This allows a single processing node to be used for any number of jobs. The Distributed Queue worker blocks until all sub-tasks are completed which means there must be at least one more worker available or the entire processing halts. The Distributed Scheduler is the best solution for job scheduling.

Within DST the development of a High Performance Computing (HPC) centre will have an impact on how future distribution work will be handled. The HPC may have its own method of communication between worker nodes. Currently both implementations of distributed processing will be able to use the extra processing power of the HPC. The Distributed Scheduler may be able to better utilise the available the resource by creating new processing resources as needed.

7. CONCLUSION

The need for a large pool of processing power and easy automation of data generation has existed as long as DST has been using detailed land combat simulations. Several bespoke solutions of distributing the processing have been created by DST but were abandoned when either the target simulation stopped being used or the software was found to not be reliable. The experience with the needs of distributed processing allowed DST to specify the requirements for their preferred solution. Working with Deakin University scheduling new approach to distributing the simulation tasks was developed. At the same time immediate work requirements saw a continuation of an internal DST implementation that also produced a functional system.

The two developed distribution systems provided a robust method for distributing a variety of processing tasks for parallel execution. The method of each framework varied. The DST developed solution used a bespoke task submission service using the JMS as the message transport to an ad-hoc network. The Deakin developed solution used a range of well support Open Source components in a fixed network. In both cases the distribution frameworks strengthen our processing capability by using existing capabilities without requiring software changes. Ultimately the requirement to execute Windows only applications saw the DST Distributed Queue solution selected as the framework to use for distributed processing.

The parallel distribution of work tasks has seen a significant benefit for data generation within DST. Without it the required simulation data or execution of the combat simulations could not be performed in a time frame that would be useful for providing valid advice. The development of the distributed processing frameworks has allowed DST to provide timely advice and to be agile to changes in data generation.

REFERENCES

Deakin, N (2013) Java Message Service API; Available from http://download.oracle.com/otn-pub/jcp/jms-2_0-fr-eval-spec/JMS20.pdf

DSTIL (26/7/2017); Virgil – A distributed computing platform, Available from https://dstil.github.io/virgil/

Holden, L., Shine, D., Dexter, R. (2016) SimR: Automating Combat Simulation Database Generation, Springer Proceedings.

Oracle (26/7/2017), Java Remote Method Invocation - Distributed Computing for Java; Available from http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html

Shine DR, Dexter RM, Russack SJ (2007) CAEn System Data User Guide v9.2. Defence Science and Technology Organisation DSTO-GD-0496