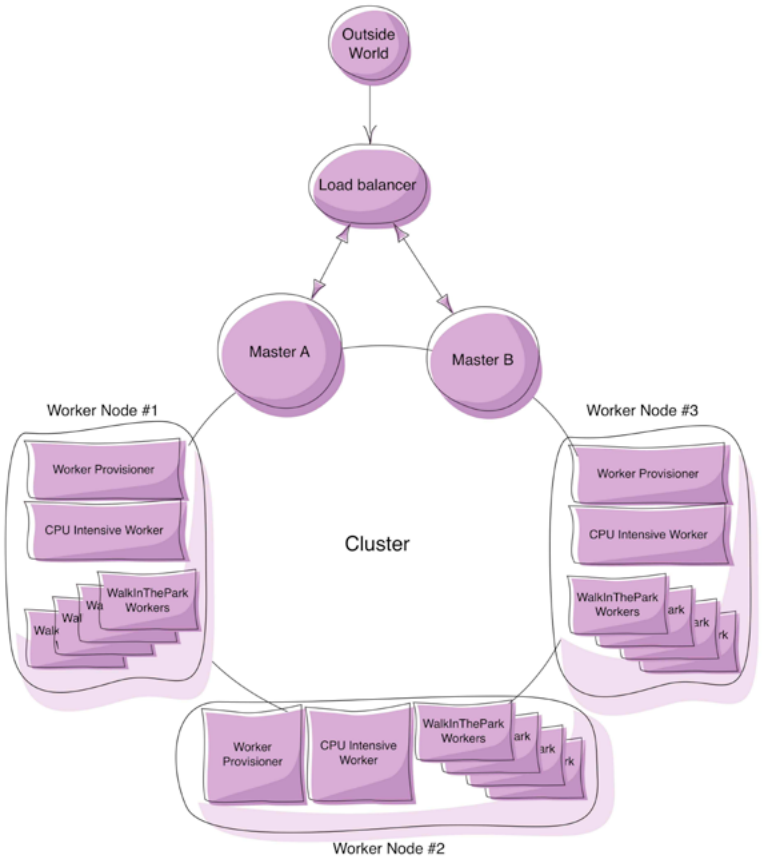
Technical Overview

Devised to perform financial modelling and simulation calculations, this system allows for highly computational tasks to be quickly distributed across cluster nodes in a resilient and scalable manner. Designed and developed using the event-based Actor[1,2] messaging development paradigm, component failure is anticipated such that failed nodes (or tasks within a node) are automatically detected and any jobs being processed at the time of failure can be resubmitted to either a healthy node or as a new task within a node. Written in Scala and running on the Java JVM means that existing Java libraries and APIs can be easily incorporated just by referencing jar libraries on the classpath.

- Designed to perform highly computational calculations (eg, modelling, simulations and realtime analytics) by distributing work tasks across cluster nodes
- Multiple data store options (Oracle, MySQL, MS SQL, Cassandra)
- Designed around the Actor and Let-It-Crash paradigms with inbuilt supervision and (re)launching of worker tasks
- A dedicated node service ensures cluster nodes are optimally utilised (eg CPU cores and memory) by assigning specific tasks to specific nodes
- Connectivity with Apache Camel for numerous enterprise integration options
- JVM based (written in Scala) allowing for seamless integration with Java libraries and APIs

Starting from a RESTful API interface, client systems can submit processing requests that are first load balanced to various ***master*** processors. These masters are responsible for splitting up and distributing the incoming job into smaller sub-tasks before forwarding to other ***worker*** processors. The payloads of these sub-tasks contain either the data to be processed or, for large datasets, details from where the data can be sourced. The sub-tasks are distributed to individual actor based worker components that are then responsible for performing the required computations. Individual workers can be categorised to help facilitate the different types of processing tasks that might be required on various nodes.



Example cluster setup

Within a node a dedicated actor process is responsible for provisioning the right number (of the right category) of workers given the incoming requests. This ***provisioner*** takes into consideration the available resources on the node (ie CPU cores and memory) before deciding whether the node can accommodate more workers and

hence tasks. This allows for both protecting individual nodes from being overloaded and also to dynamically adjust the worker categories on a given node as requests change. After processing, an ***aggregator*** is responsible for collating the results from individual nodes and, where necessary according to business logic, resubmitting tasks for further processing. When further processing is no longer required, the final results are either stored or passed to downstream systems. (In the event of the results being stored, an optional notification can be sent via a websocket to the initiating system - eg a web client or Excel spreadsheet).

- RESTful API for ease of both client and developer interaction
- Tested on both cloud (AWS) and inhouse (OpenStack) cluster installations
- Secure Web Socket interface for server to client push (messaging / notifications etc)
- Configurable load balancing of incoming client requests (round-robin, least-connected & ip-hash)
- Work 'tasks' are scattered across nodes for processing before being gathered together after processing is complete[3]
- Utilises the enterprise standard Ehcache for caching
- Client username / password or OAuth for client authentication
- Results requiring further computations can be easily resubmitted as repeat processing jobs
- Optional Computational Statistics Module

Planned :

- Active Directory integration
- Dynamically adjust cluster size as workload increases and decreases[4]

Testing to date has included :

FX trading simulation

A 9 node Amazon AWS cluster containing 1 master and 32 workers was deployed in a virtual private cloud. Over 47 million FX tick quotes were analysed for trading signals based on a predefined algorithmic trading rule. The simulation took just over 4 minutes to complete and this time included, file I/O, main trade processing, generating trade statistics and persisting the results to a database outside AWS.

(full details on the following page).

For more details and further information please contact enquiries@anglowide.com

Testing details

- FX trading simulation

  Overview

  An algorithmic trading rule was defined that looked for a series of price events over the previous 'n' minutes. If the events happened then an FX trade would be opened. The trade would then be closed at pre-defined stop/loss or profit levels. The sample data set contained 47.6 million €/$ quotes (between Jan 2012 and May 2014) and stored in multiple files on Amazon AWS S3. A cluster consisting of 1 master and 8 worker nodes was created within an AWS virtual private cloud (VPC). Worker nodes had 4 virtual CPUs so they were configured to allow a maximum of 4 workers each, resulting in a total of 32 available workers across the cluster. A single simulation request was sent to the master node which then split the request (based on calendar month) and distributed the work to all workers. Once all workers had finished processing their quotes, an aggregator service checked to see if any jobs needed resubmission (due to logic within the trading rule) and requested some jobs to be re-run before finally saving the results and informing the master node.

  Results (all in seconds)

  2 : Master requesting 32 workers and all workers responding
  21 : File I/O (loading quote files)[5]
  185 : Main processing : Looking for trades to open and then close[5]
  11 : Performing a re-run on a subset
  25 : Generating trade statistics and saving results in a database.

  244 : Total : From initial request to final results

  AWS VPC cluster details (all unoptimised and running Ubuntu 12.04 LTS + Java 1.6) :
  Master node : 1 x m3.large (2 x vCPU Xeon E5-2670, 7.5 GB RAM)
  Worker nodes : 8 x c3.xlarge (4 x vCPU Xeon E5-2680, 7.5 GB RAM)

For more details and further information please contact enquiries@anglowide.com

[1] http://en.wikipedia.org/wiki/Actor_model
[2] http://architects.dzone.com/articles/increasing-system-robustness
[3] Similar to the Scatter-Gather EIP (http://www.eaipatterns.com/BroadcastAggregate.html)
[4] When using either cloud hosted or locally hosted IaaS components (eg AWS or OpenStack)
[5] Average across all worker nodes