Engineering Applications of Artificial Intelligence 24 (2011) 1120–1127

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

A multi-agent simulation framework on small Hadoop cluster

Prashant Sethia, Kamalakar Karlapalem *

Centre for Data Engineering, International Institute of Information Technology, Hyderabad, India

ARTICLE INFO

Available online 23 July 2011

Keywords:
Multi-agent simulation
Design
Experimentation
Reliability
Cloud computing
Fault-tolerance
Failure-resilience
Agent architecture

ABSTRACT

In this paper, we explore the benefits and possibilities about the implementation of multi-agents simulation framework on a Hadoop cloud. Scalability, fault-tolerance and failure-recovery have always been a challenge for a distributed systems application developer. The highly efficient fault tolerant nature of Hadoop, flexibility to include more systems on the fly, efficient load balancing and the platform-independent Java are useful features for development of any distributed simulation. In this paper, we propose a framework for agent simulation environment built on Hadoop cloud. Specifically, we show how agents are represented, how agents do their computation and communication, and how agents are mapped to datanodes. Further, we demonstrate that even if some of the systems fail in the distributed setup, Hadoop automatically rebalances the work load on remaining systems and the simulation continues. We present some performance results on this environment for a few example scenarios.

1. Introduction

Multi-agent simulation is an important research field in today's scenario and analyzing emergent behaviors in such simulations largely depend on the number of agents involved. More the number of agents involved with detailing of agent decision making and communication, closer is the result obtained to the real world. A classic example would be simulating traffic of a city with millions of human agents and thousands of commuting agents like trains, buses. Due to large number of agents, time taken for such simulations becomes large and so we resort to a distributed computing solution.

In simulations involving millions of agents, the running time for each simulation cycle can be of several seconds or even minutes; and when run for a large number of cycles, the total simulation time can be of several hours or days. If some of the machines fail during the run-time, then the entire simulation needs to be restarted. If we can somehow dynamically re-balance the work load on the remaining number of machines and maintain logs of the simulation progress, we can continue the simulation from the point of failure. So, we require a fault-tolerant and failure-resilient framework which is also easily extensible to run on a large number of processors and agents.

Hadoop (http://wiki.apache.org/hadoop) is a promising option in this respect. It takes care of non-functional requirements, like scalability, fault-tolerance, load balancing, and the framework developer only needs to develop a layer for agent-based simulation on top of it. If some systems in the distributed environment fail, the simulation does not stop. Hadoop automatically rebalances the work load on remaining systems and continues to run the simulation. Further, Hadoop facilitates dynamic addition of new nodes in a running simulation. In this paper, we present a design of an agent-based simulation framework implemented on Hadoop cloud. Being developed on top of Hadoop, it inherits Hadoop's afore-mentioned advantages.

1.1. Related work

Developing tools for multi-agent simulations has always been an active area of research (Railsback et al., 2006), with emphasis being laid on different aspects – architecture, scalability, efficiency, fault-tolerance and effectiveness of the system. A number of frameworks have been developed such as Netlogo (Tisue, 2004), ZASE (Yamamoto et al., 2007), DMASF (Rao et al., 2007) and MASON (Luke et al., 2005).

Tisue (2004) introduced a novel programming model for implementing agent-based simulations, which eased the development of complex agent-models and scenarios. It manages all the agents in a *single thread* of execution, switching execution between different agents, deterministically and not randomly, after each agent has done some minimal amount of work (simulated parallelism). The simulated parallelism provides deterministic reproducibility of the agent-based simulation every time it is run with same seed for random number generator; which was one of the implementation goals of Netlogo. Further, a visualization module provides 2D/3D

* Corresponding author.
E-mail addresses: prashant.sethia@research.iiit.ac.in (P. Sethia), kamal@iiit.ac.in (K. Karlapalem).

0952-1976/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.engappai.2011.06.009

visuals of the ongoing simulation. However, Netlogo is not able to distribute the computation on a cluster of computers and hence is not scalable.

MASON (Luke et al., 2005), developed in Java, provides a platform for running massive simulations over a cluster of computers. It has a layered architecture with separate layers for agent-modeling and visualization, which makes decoupling the visualization part easier. It has the capability to support millions of agents (without visualization). Checkpoints of agent data are created on disk for offline visualization.

ZASE (Yamamoto et al., 2007) (developed in Java) is another scalable platform for running billions of agents. It divides the simulation into several smaller agent-runtimes, with each runtime controlling hundreds of thousands of agents and running on a separate machine. It keeps all agents in main memory without the need to access the disk. A thread-pool architecture is followed with several agents sharing a single thread of execution.

DMASF (Rao et al., 2007) (developed in Python) has an architecture similar to MASON and ZASE. Like ZASE, it divides simulation into several smaller runtimes executing on different computers and same thread is shared by several agents. But it uses MySQL database for providing scalability with the help of secondary storage rather than getting bounded by the limited main memory. Similar to MASON, it has a modular architecture separating agent-modeling and visualization. Further, it dynamically balances the agent execution load on different machines.

However, the ability to handle hardware failures is lacking in all the three (MASON, ZASE, DMASF). If some of the systems using these frameworks fail during the simulation run, then the simulation needs to be restarted from the beginning.

SWARM (Minar et al., 1996), RePast (Collier, 2001) and JAS (Sonnessa, 2003) are some of the other widely used frameworks for studying emergent agent-behaviors through agent-based social simulations. However, they lack the capability to manage more than one system and hence are not scalable.

1.2. Contribution and organization

Our proposed framework developed on Hadoop provides three major advances to the current state of art: (i) Dynamic addition of new computing nodes while the simulation is running; (ii) Handling node failures without affecting the ongoing simulation by redistributing the failed tasks on working systems; (iii) Allowing simulations to run on machines running different operating systems. Further, the framework incorporates several optimization techniques: (i) clustering of frequently communicating agents (for reducing inter-processor communication); (ii) caching of results (for improving performance) that are run on Hadoop cloud.

Section 2 presents the architecture of Hadoop. Section 3 gives the framework architecture built on top of Hadoop. Issues faced in developing the framework and our proposed solutions for overcoming them are presented in Section 4. In Section 5, we present some performance results for few example scenarios. Finally, in Section 7 we present some conclusions.

2. Hadoop architecture and map-reduce model

Hadoop (http://wiki.apache.org/hadoop) is an Apache project which develops open-source software for reliable and scalable distributed computing. It maintains a distributed file system, Hadoop Distributed File System (HDFS) (http://hadoop.apache.org/hdfs/) for data storage and processing. Hadoop uses classic *Map-Reduce* programming paradigm to process data. This paradigm easily fits a large number of problems (http://code.google.com/edu/parallel/mapreduce-tutorial.html). Hadoop consists of a single master system (known as *namenode*) along with several slave systems (known as *datanodes*). For failure resilience purposes, it has a *secondary namenode* which replicates the data of *namenode* at regular intervals.

2.1. Hadoop distributed file system (HDFS)

HDFS (http://hadoop.apache.org/hdfs/) is a block-structured file system: individual files are broken into blocks of a fixed size (default size is 64 MB), which are distributed across a cluster of one or more machines (*datanodes*); thus all the blocks of a single file may not be stored on the same machine. Thus, access to a file may require access to multiple machines, in which case a file could be rendered unavailable by the loss of any one of those machines. HDFS solves this problem by replicating each block across a number of machines (three, by default). The metadata information consists of division of the files into blocks and the distribution of these blocks on different *datanodes*. This metadata information is stored on *namenode*.

2.2. Map-reduce paradigm

The *MapReduce* paradigm transforms a list of *(key, value)* pairs into a list of values. The transformation is done using two functions: *Map* and *Reduce*. *Map* function takes an input *(key1, value1)* pair and produces a set of intermediate *(key2, value2)* pairs. The *Map* output can have multiple entries with the same *key2*. The *MapReduce* framework sorts the *Map* output according to intermediate *key2* and groups together all intermediate *value2*'s associated with the same intermediate *key2*. The *Reduce* function accepts an intermediate *key2* and the set of corresponding *value2*'s for that *key2*, and produces one or more output *value3*'s.

(i) `map(key1,value1) -> list 〈(key2,value2)〉`
(ii) `reduce(key2, list 〈value2 〉) -> list 〈value3 〉`

The intermediate values are supplied to the *Reduce* function via an iterator (http://download.oracle.com/javase/6/docs/api/java/util/Iterator.html). This allows handling lists of values that are too large to fit in memory. The *MapReduce* framework calls the *Reduce* function once for each unique *key* in sorted order. Due to this the final output list generated by the framework is sorted according to the *key* of *Reduce* function.

For example, consider the standard problem of counting the number of occurrences of each word in a large collection of documents (Dean and Ghemawat, 2004) (Table 1). This problem can be solved by using the *Map* and *Reduce* functions. The *Map* function emits each word along with an associated count of occurrences (just '1' in this simple example). The *Reduce* function sums together all counts emitted for a particular word.

Table 1
Map-reduce solution for word-count as it occurs in Dean and Ghemawat (2004).

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result=0;
  for each v in values:
  result += ParseInt(v);
  Emit(AsString(result));
```