Distributed Simulation of Distributed Systems

Authors	
Alex Boyko	o.y.boyko@student.vu.nl
Joshua Kenyon	j.m.kenyon@vu.nl
Geoffrey Frankhuizen	g.p.s.frankhuizen@vu.nl

Support Cast
Laurens Versluis, Fabian Mastenbroek, Alexandru Iosup, and Animesh Trivedi

Abstract

This paper investigates how to create a distributed version of the event simulator known as OpenDC, which currently runs on a single machine. The system we have made to handle this problem uses nodes laid out in a ring structure that use message passage to communicate with each other. We ran experiments to test our system with an increasing number of instances and an increasing number of messages. The overhead introduced by the message passing negatively affected the speedup. The main result we discovered was that although we solved the single machine RAM bottleneck, the problem has shifted to a speedup bottleneck.

1 Introduction

The problem we are facing is a single-machine simulator with a RAM bottleneck. A single machine only has so much RAM, therefore limiting the amount of things that can be simulated. A distributed version of this simulator could overcome this problem by spreading the workload out over multiple machines.

The existing system is OpenDC [1], an event simulator that currently runs on a single machine. Related work to this system is a way to do event simulation in parallel, namely parallel and discrete event simulation [2]. Some ways to tackle parallel and discrete event simulation are conservative time synchronization [3] and optimistic time synchronization [4].

We aim to implement a distributed version of OpenDC, to tackle the RAM bottleneck of the current implementation. The focus of our system is to have multiple instances of the simulator working in parallel, with message passing and balanced workloads.

The strucure of our paper is as follows: Section 2 describes the background on the application, while Section 3 is our system design. Section 4 shows our experimental results, and Section 5 discusses them. We end with Section 6 with our conclusion.

2 Background on the application

2.1 OpenDC

OpenDC is the system we were initially working with. It is a simulator of distributed systems. Currently, its implementation only uses a single instance. The bottleneck of this implementation is the RAM usage, which limits the number of virtual machines and the size of the workload. To overcome this workload, we intended to alter the implementation to be able to run on multiple instances. This way the virtual machines and workload can be distributed over multiple machines. Message passing and processing are important parts of the simulation. These operations have to happen in a time consistent manner. OpenDC can be an important tool for researching large-scale distributed machines, as it allows for running simulations on far smaller systems.

2.2 Functional Requirements

The functional requirements of our system are the following:

FR1 - Scalability. The system must be able to handle a growing amount of work by adding resources to the system.
FR2 - Load balancing. The workload must be balanced across the different instances of the simulator.
FR3 - Reliability. The system must not lose or corrupt data.
FR4 - Performance. The system must be able to process bigger workloads and hold more virtual machines on multiple instances of the simulator compared to a single instance.

2.3 Non-Functional Requirements

The non-functional requirements of our system are the following:

NFR1 - If we double the amount of simulator nodes in our system, it should be able to handle at least 50% more work.
NFR2 - The system can continue to run as long as 90% of the simulator nodes are running.

3 System Design

3.1 System Operation

The system is compromised of one or more nodes, which are laid out in a ring structure. Each node is connected with two of its neighbors (assuming we have three or more nodes), one to which it can send envelopes, and the other from which it can receive them.

A node passes envelopes on the ring. Envelopes always have a destination. The payload of the envelope is one of three different types: message which contains a message from the simulation, confirmation which is an acknowledgement of a message, or update which contains state information of a node.

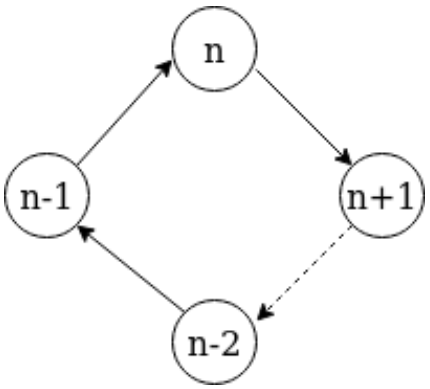

Figure 1: An overview of the system's architecture. The nodes in the system are laid out in a typical ring structure.

The message is simulation specific. Each message from the simulation has three components: a payload, a timestamp, and a destination. Timestamps are used to pass the messages in a time-consistent manner. We take the destination modulo the number of nodes to determine which node in the ring structure the message should be sent to.

Each node has a queue with messages which are ordered by timestamp in ascending order. Until this queue is empty, the node will check to see if its message has the lowest timestamp of all messages that still need to be sent. For this, the node keeps a table with the lowest timestamp and whether the node is done for each node. An example of such a table can be seen in Table 1. If the node has the lowest timestamp of the nodes that are not done, then it can send this message. In the case where two nodes have the same timestamp, the node with the lowest ID takes precedence.

ID	Timestamp	Done
0	3	false
1	1	true
2	5	false
3	2	false

Table 1: A table showing the state information of the nodes in the system from the eyes of a single node. In this case the node with id 3 can send its message, since it has the lowest timestamp of the nodes that are not done.

Once this message is sent it will be passed through the ring to the destination node. After receiving the message, it is processed and a confirmation is sent back to the sender. Once the confirmation is received, the original sender sets its timestamp to the next message in its queue or sets its *done* flag to true. Every time a node updates its timestamp or *done* flag it broadcasts its state information over the ring.

Once all the nodes have have set their *done* flag set to true and the state information is propagated, the nodes processes are terminated.

3.2 Scalability

In this system each node is only assigned a part of the workload. This means that the amount of RAM used per node decreases with the number of nodes for a fixed workload. Thus, the system can handle bigger workloads when introducing additional nodes.

3.3 Additional Feature: Repeatability

The system is deterministic, the messages are always sent and processed in the same order. Additionally, the messages are generated based on parameters of the setup, the number of nodes, and a fixed seed. This means that anyone with access to the source code [5] can reproduce our results concerning consistency and scalability.

4 Experimental Results

4.1 Experimental Setup

Our implementation is written in Java and uses the built-in library for RMI. To run our experiments, we have written a Bash script. We ran the experiments on a node in the DAS-5 [6]. This node has two Intel Xeon E5-2630 cpus with 16 threads each and 128GB of RAM. Our implementation ran in OpenJDK build 1.8.0_161-b14. We introduced a central server used for testing the consistency and measuring the time the system ran per workload. We measure the real time from the moment all the messages have been generated until the central server has received a message from each node that it is done. Each message has a 1KB payload.

4.2 Experiment 1

Our first experiment tested the change in speedup when running multiple instances of the system. We fixed the number of messages to 100000. We measured the speedup for 2, 4, 8 and 16 instances. The results are shown in Figure 2.

We see that, when running the system with multiple instance, the speedup is always smaller than one and strictly decreases as more instances are introduced. This is likely due to the increased overhead for each introduced instance. This overhead comes in the form of the number of envelopes that need to be passed. For a message that is sent over the ring a remote invocation has to be made for each node in the ring, the same holds for an individual update of a node's state. However, for each introduced instance, an additional update needs to be sent. This means that the number of these type of messages are in $O(\#instances^2)$.

4.3 Experiment 2

Our second experiment tested the change in speedup for different amounts of messages. We fixed the number of instances to two. We measured the speedup for 100, 1000, 10000, 100000 and 1000000 messages. The results are shown in Figure 3.

The speedup of two instances was generally below one, which the workloads of 100 and 1000 being the exception. The speedups above one can be explained by the variance introduced by

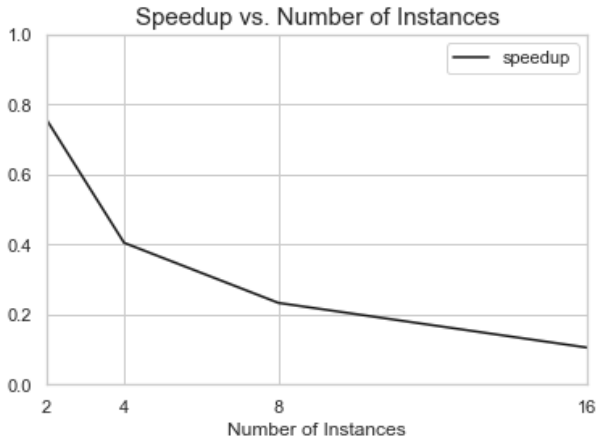

Figure 2: A graph showing the speedup versus the number of instances in the system. The average measured times for the number of instances are: 1 - 23.141s, 2 - 30.656s, 4 - 57.181s, 8 - 99.285s and 16 - 219.542s.

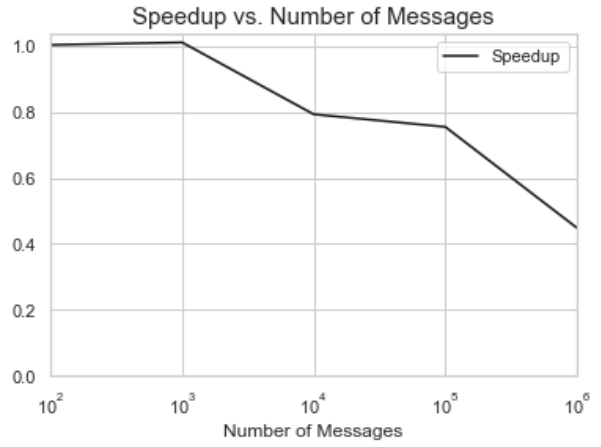

Figure 3: A graph showing the speedup versus the number of messages in the workload. The average measured times for the number of messages with one instance are: 100 - 1.082, 1000 - 1.617, 10000 - 4.09, 100000 - 23.141, 1000000 - 201.153. The average measured times for the number of messages with two instances are: 100 - 1.078, 1000 - 1.599, 10000 - 5.158, 100000 - 30.656, 1000000 - 448.704.

the short run times of the jobs, 100 or 1000 messages is not a big workload for this system. The cases where there are more messages seem to indicate the general trend. The way the system is implemented does not allow for messages to be sent or processed in parallel. This puts an effective limit of one on the speedup that can be gained. Additionally, a significant overhead in the form of message passing is present when multiple instances are used. This explains the declining trend in speedup as the number of messages increase.

5 Discussion

Introducing additional instances can be used to resolve the RAM bottleneck, which means that bigger workloads can be processed. However, the results of our experiments show that we had speedups below one both when increasing the number of instances in the system and when increasing the number of messages. This means that we have a trade-off between RAM and performance. The problem has been shifted.

We believe that the system proposed in this paper should not be used to implement message passing for OpenDC, as there is too much of a performance overhead. To make the system more suitable the ring structure would need to be altered in order to reduce the number of messages that need to be sent. For instance, the nodes could be connected to a few nodes that are far away to reduce the number of hops a message needs to travel. The processing of the messages could also be done in parallel to overlap computation and communication.

If we would increase the number of messages by an order of magnitude in our experiment, then we expect the speedup to drop even further below 0.5 for two instances. If we increase the number of instances in the system by an order of magnitude, then the speedup will likely drop to below 0.1.

6 Conclusion

We have introduced a distributed system for time consistent message passing. The systems' architecture is a ring structure. This system gave us insight in how to solve the RAM bottleneck for the non distributed version of OpenDC. However, our experiments showed that the bottleneck has been shifted to speedup. This bottleneck can be explained by the overhead introduced by a ring structure. This overhead can be reduced by adding complexity to the ring. Further research is needed to resolve the new speedup bottleneck for this system to be a viable component in OpenDC.

References

[1] "OpenDC, Collaborative Datacenter Simulation and Exploration for Everybody." `https://github.com/atlarge-research/opendc-simulator/`. Accessed: 19/12/2019.

[2] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, pp. 30–53, Oct. 1990.

[3] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. Softw. Eng.*, vol. 5, pp. 440–452, Sept. 1979.

[4] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 404–425, July 1985.

[5] "Time Consistent Message Passing." `https://github.com/GreenGearX/Time-Consistent-Message-Passing`. Accessed: 19/12/2019.

[6] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, pp. 54–63, May 2016.

7 Appendix A: Time sheet

Type of Activity	Time
Think-time	40 hours
Dev-time	34 hours
Xp-time	4 hours
Analysis-time	4 hours
Write time	21 hours
Wasted time	33 hours
Total-time	**136 hours**

Table 2: The amount of time that was spent on each activity.