2012

Distributed Conway's Game of Life Simulation

SFU CMPT 431 Final Report

Group Members

Chang Lu	#301151627	luchangl@sfu.ca
Yuke Zhu	#301183470	yukez@sfu.ca
Yao Xie	#301183473	xieyaox@sfu.ca
Yang Liu	#301183499	yla269@sfu.ca
Xiaying Peng	#301133692	xiayingp@sfu.ca

2012 Spring

2012/4/10

MENU

INTRODUCTION

Project Overview

The Game of Life is a visualized cellular automaton devised by the British mathematician John Horton Conway in 1970. It was popularized when it was first mentioned in an article by Martin Gardner [1] published on Scientific American. The universe (board) of the Game of Life is a two-dimensional orthogonal grid of square cells. Each cell has two states, alive and dead. [2] Each cell has eight neighbors, four adjacent orthogonally, four adjacent diagonally. The transitions occur based on the number of neighbors at each step (also called generation). The Conway's genetic laws are simple: [2]

Survivals	Every counter with 2 or 3 neighboring counters survives for the next generation.
Deaths	Each cell with 4 or more neighbors dies (is removed) from overpopulation. Every counter with one neighbor or none dies from isolation.
Births	Each empty cell adjacent to exactly 3 neighbors is a birth cell. A counter is placed on it at the next move.

There is an initial state for each cell. The initial pattern is often called start configuration. All the transitions occur simultaneously for each generation, not sequentially. Therefore, the board is updated as a whole. Michael J Young discussed some typical uses of cellular automata in his paper [3]. With delightfully simple rules, cellular automata can be used in modeling and studying natural system, including life itself.

According to the features of Conway's Game of Life, the universe can be divided into smaller parts and computed separately, given the boundary cell states. Therefore, a distributed computing architecture is well-suited for efficient computing large-scale automata. In this paper, we discussed our approach to implementing distributed Conway's Game of Life simulation on over hundreds of computers. The cellular automaton model with huge amount of cells can be further used to study complex systems that can be modeled by a large number of simple elements that with local interaction only, such as fluid dynamics and road traffic simulation.

Objectives

- Implement a distributed Conway's Game of Life computational simulation
- Handle large-size packetized network messages
- Build a graphical interface to visualize the universe
- Support over 100,000,000 cells and 10,000,000 lives
- Support dynamically adding and deleting computing nodes
- Finish one cycle with 100,000,000 cells within a quarter of second

Challenges

- Computing nodes must exchange information to coordinate with each other
- Various messages must be designed to implement dynamically adding and deleting nodes
- Server must take control to guarantee transitions occur simultaneously
- Network messages should be minimized in order to keep the system from network-bound with only a few computing nodes (clients)

Related Work

Some memorized algorithm, like Hashlife [4], can be used to compute long-term fate of a given starting configuration; some significant drawbacks limit the usage of such algorithms. One of the most obvious drawbacks is the huge memory consumption, especially for patterns with large entropy. Besides, cellular automata which run on single machines do not have good scalability, which will be overwhelmed by very large configurations. Due to the constraints of standalone automata, new architectures are implemented to provide environments based on the cellular automaton theory. Some significant examples are CAM [5] and CAPE [6]. In Domenico Talia's paper [7], more examples of simulation environments based on cellular automata are introduced.

However, the environments are complicated and often require a bunch of additional codes to run a simulation. In contrast with developing a complicated simulation environment, our project aims at implementing a simple Conway's Game of Life automata, which can be easily deployed in hundreds of computers. Hopefully, the system can overcome the limitation of standalone cellular automata and thus achieve a high scalability.

OVERVIEW OF ARCHITECTURE

Initial Architecture Scheme

Our initial architecture scheme is a balanced tree, where the leaf nodes are clients, and the other nodes are sub-servers. All computation is completed by clients, while message exchanging, structure maintaining and synchronization are sub-server's responsibilities. The general work cycle should be like this:

Server sends a start command; sub-servers pass the command together with necessary border information down to clients. After one client completes computing, it reports to sub-servers; then, sub-servers pass the results to the server. If the tree structure needs to be adjusted, when a client comes or leaves, server will ask sub-servers to help rearrange the tree structure in order to make sure that the tree is balanced.

The advantages of this architecture scheme are as follow:

- The tree is balanced, so that the distribution is fair.
- No clients would be assigned more than two times workload than others.
- It is easy for the parent to control its children.

Parents split theri work into two and assign half to left children, the other half to right children. It doesn't care whether the children assign the work to their children. When requesting, it will just ask their children to report.

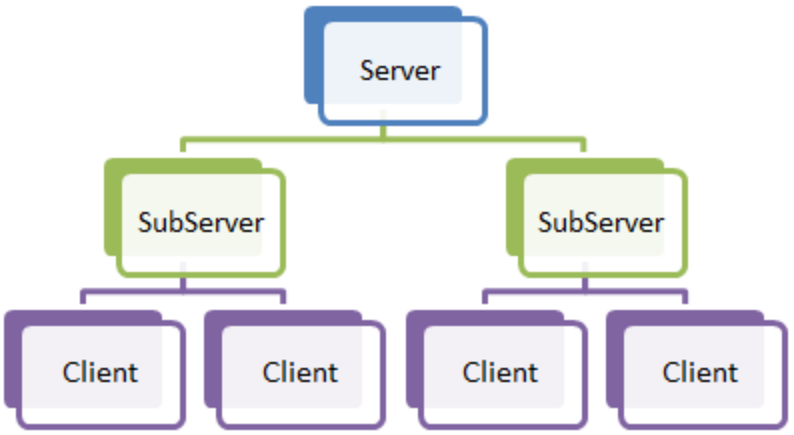


Fig. 1 Initial architecture scheme

The major disadvantage is obvious. Only half of the nodes are actually doing computing work. The other half will not be involved in computation, which is a large waste of resources. Besides, it consumes large network traffic when the requests and reports have to go through the hierarchy between top and bottom.

In this structure, the number of messages will be proportional to the depth of the tree. That may cause the significant network burden and delay. Assuming the messages are sent to the server directly, it will greatly reduce the number of required messages. Therefore, a revised (and better) structure comes out.

Revised Architecture Scheme

In this revised structure, the server does the work as the sub servers do in the previous scheme. In this way, we can avoid the network messages passing along the tree. Thus, no resources are wasted, since all nodes work as computing nodes (clients).

However, it would be more complex than the previous scheme. It does not allow us to have the simple parent to children distribution any more. That's the tradeoff for lower network load and faster responding time.

Now, the server has to maintain the tree structure, send request, get report and deal with the information exchange between clients, which gives much pressure to the server. So we make client communicate with each other to get all information they need in computation.

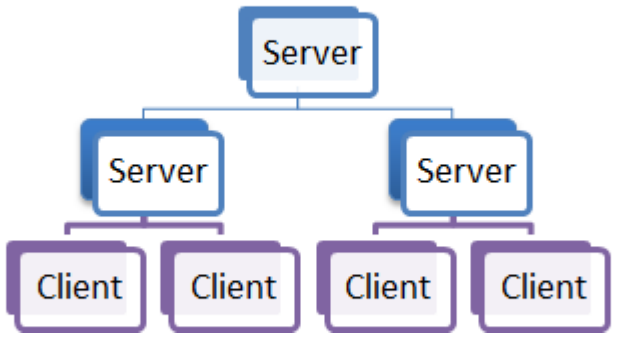


Fig. 2.1 All sub-servers are inside the server itself

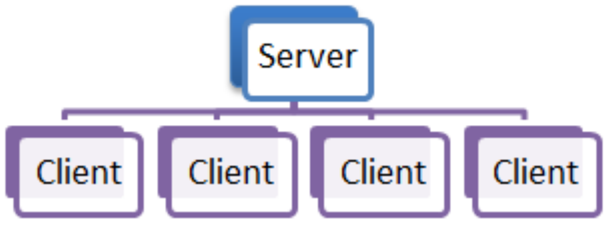


Fig. 2.2 Actual architecture

Physically, the tree structure does not exist. However, the server still maintains such a tree to make sure the distribution is fair. In the following analysis, we will still use this scheme with tree structure to explain the maintenance in a clearer manner.

DETAILS OF THE DESIGN

Assumptions

Our whole system is based on the following assumptions:

- **The network is reliable**, which means two nodes are guaranteed to be connected. Although packages may drop or corrupt, the messages are guaranteed to be transmitted correctly by TCP channel.
- **All clients will not quit unexpectedly.** Before leaving the system, it will send a leave request to server and wait for permission from server.

Adding Procedure

When a new client wants to join the distributed system, server will first find it a pair. Our adding schema is similar to the procedure of adding a node to a binary heap. The adding operation in heap is simply to append the new node to last position. Here is the example to show the change in the tree structure after adding a new node:

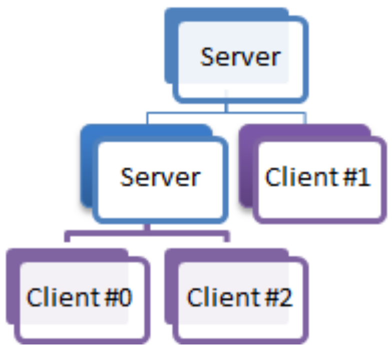


Fig. 3.1 Before adding a node

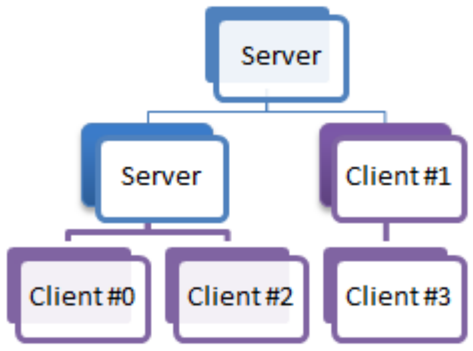


Fig. 3.2 After adding a node

Fig 3.1 and 3.2 show us the change after adding a new node. Notice that it violates our design, since Client #1 is not a leaf node any more. So the server adjusts by simply putting client1 down one level to make it a sibling of Client #3, the new coming node. Finally, we get a balanced tree again.

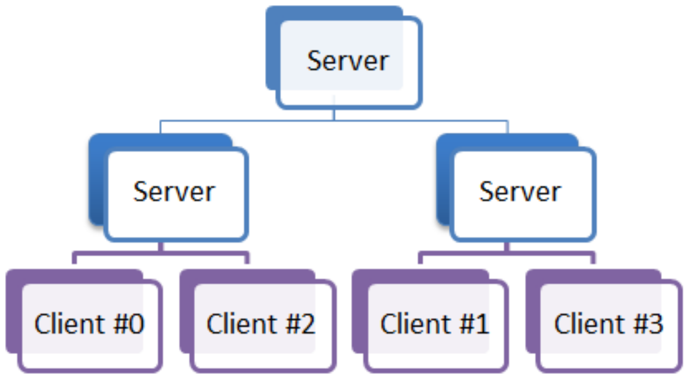


Fig. 4 New tree structure after rearranging

This pulling down operation will not cause the tree depth to increase, so it will not break the validity of the tree. The adding operation is completed.

Leaving Procedure

Deleting is in practice the reversed operation of adding. Consider a heap deletion first. The heap idea is to use the last node to substitute the deleted position and adjust the structure later. So our method performs in a similar way.

Let's consider the structure. If Client #2 asks to leave, the server uses the last node, Client #3, to substitute its position.

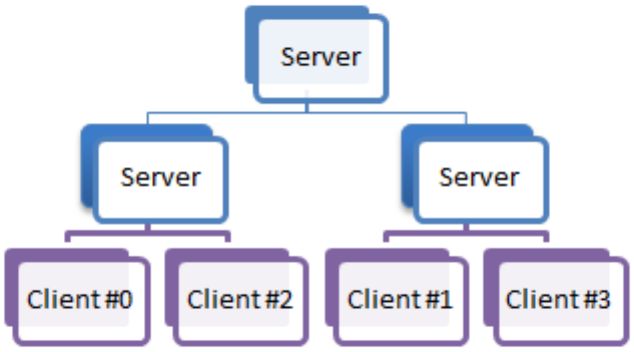


Fig. 5.1 Tree structure before deleting

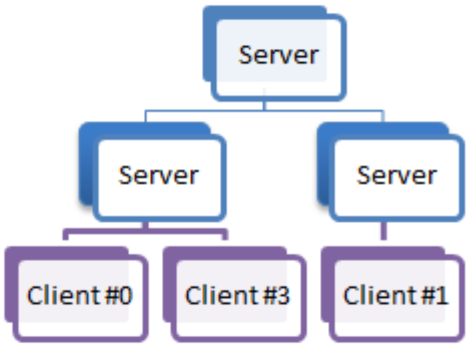


Fig. 5.2 Tree structure after deleting

And then, the server needs to do some rearrangement to avoid violation to our design. Client #1 no longer has pair now, so it will be pulled up one level (just like a heap).

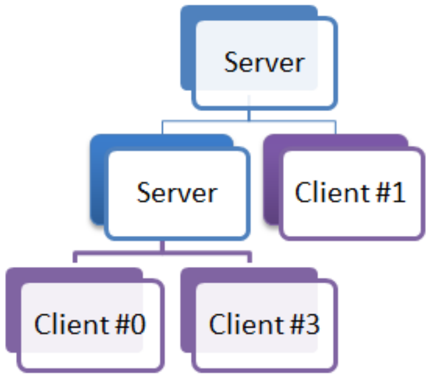


Fig. 6 New tree structure after rearranging

IMPLEMENTATION

Technical Research

Programming Languages and Libraries

At first, we used Java and wrote a simple demo in Java. Java provides build-in serializable methods and user interface methods. However, Java has relatively low performance compared with C and C++.

To improve the performance, we tried C++ with ZeroMQ and Google Protocol Buffers and used Qt for the graphic user interface. After spending some time researching, we found several limitations: first, it is hard for

Qt to work with ZeroMQ; second, we were not allowed to install the dynamic libraries in CSIL, which will limit our performance testing. Therefore, we finally switched back to Java.

Message Passing

Java Socket is initially used at first to build a distributed demo with one client only. But when updating it to several clients, Java Socket is not appropriate because the connection method will block the system.

Then we tried using Java message queue facilities; however, it requires J2EE instead of J2SE, which is not installed in CSIL machines. Eventually, we decided to use NIO, short for Java Non-blocking I/O. It constructs a selector to monitor all connection requests and incoming messages, and push them into a message queue. Therefore, our core system just needs to check the message queues. On the other hand, Java NIO will help construct a sending message queue.

Compression

We first used some build-in compression methods (deflator) to make messages smaller. It works fine for small data. However, it turns to be significantly slow for large data size. It costs far more time to compress a message than sending the original one. After, we tried several other compression algorithms. LZW algorithm works pretty well and achieves a compression ratio up to 40. However, there is still not much gain in time. As a result, compression is turned off in the performance testing.

Adding

Special Case

If the incoming client is the first one in the whole system, just simply add it to tree using two messages. First, a message contains board information from the server to this client. Then, a confirm message from client to server.

General Case

If the new client is not the first one, we use the following scheme. To complete an Adding operation requires four messages.

J1	A join request from client to server
J2	A command to make oldest client to split

R5	A message containing the board and neighbor information
R4	A confirmation information to show everything is done

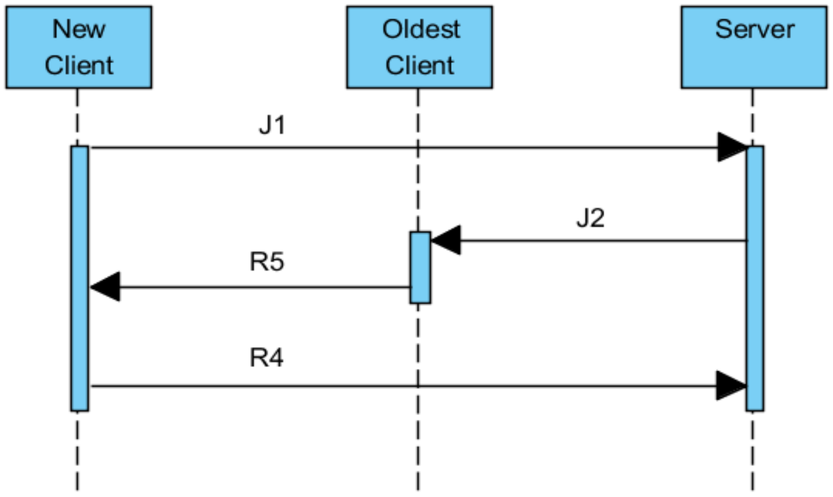


Fig. 7 Message flow for adding nodes

For more detailed information about various messages, please refer to Table 1 in Appendix I.

Leaving

Special Case

If the leaving client is the only one client, it just simply leaves. Then server will keep the current information and go idle until a new client joins. If the new leaving client is one of the last pair in the tree, server will ask the last pair merging and then allow it to leave.

General Case

If the first two cases are not satisfied, we use the following scheme to deal with the general case. A general leaving operation needs seven messages:

R3	A leaving request from client to server
L1	A command to make the last pair merge
L2	A message with board information and neighbor information, used in merging
R4	A confirm information
L3	A permission to leave
R5	A message containing board and neighbor information

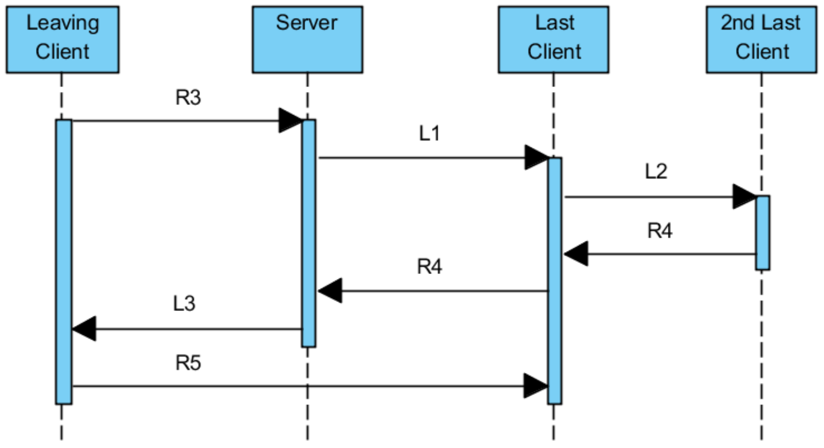


Fig. 8 Message flow for deleting nodes

To express the work flow of adding and deleting nodes formally, we designed the following state machines for both server and client.

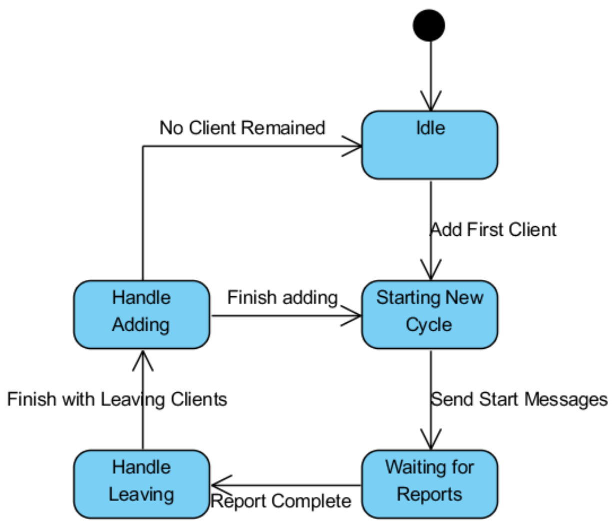


Fig.9.1 Server state machine

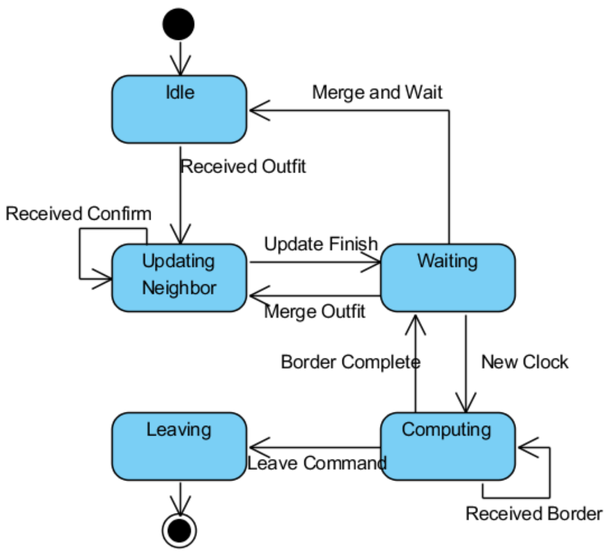


Fig. 9.2 Client state machine

Neighbor Updating

All clients keep track on its own neighbors. When a new client joins, it will be sent the neighbor information. Besides, when the tree structure changes, for example new clients come or old clients leave, it will receive a neighbor updating message and update the neighbor information. All neighbor information is stored in an array list in clockwise order, which helps clients change border information with its neighbors.

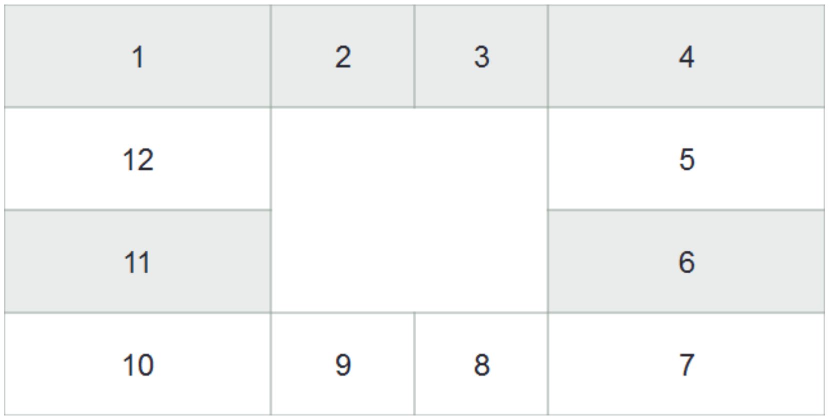


Fig. 10 Each client has 12 neighbor positions

The worst case is that, a client has 12 neighbors, each in different positions. The best case is, when the board is well distributed to 2^n clients, Position #2 and #3, #5 and #6, #8 and #9, and #11 and #12, are covered by four neighbors. In this case, a client has 8 neighbors in total. Even though, in our design, 12 neighbors is impossible in fact, the practical worst case is a client with 10 neighbors, we still use the general case to simplify the programming and support potential updates to the further design.

Network Interfaces

Since in our system, each host needs to communicate with many other hosts simultaneously. And the messages from the hosts may arrive in many possible orders due to the uncertainty of the network, for example, a client may receive a border information message before receiving the server's new cycle message, from a client that receives the new cycle message much earlier. Therefore, our implementation must handle all the possibilities. To solve this problem, we considered the following schemes:

- Each host processes the messages in a certain order that consists with the system design, wait if the right message has not been receive yet.
- Create a new thread for each connection and coordinates the threads using IPC techniques.
- Implement a message queue facility, messages from all connections are pushed into a queue, and handle possible orders in the client/server logic.

The first scheme was rejected at first because of its inefficiency. If a host must wait before receiving the message in a certain order, the network delay is certain to be amplified.

When deciding between the second and the third schemes. We found that it is much more complicated to implement our design based on the second scheme. Besides, the handling of the message order in the third scheme in program logic is not very hard. All of this made us finally choose *message queue* as our solution.

To implement the message queue facility, we utilized the non-blocking socket in Java NIO package. The package offers a mechanism that a selector runs in a thread, polling over all the connected sockets without blocking the thread, and calls the corresponding handler when there is available data in a channel.

In out implementation, each host runs a "message receiver". Whenever there is data in a channel, the receiver will read it out, parse it to message object and push the message object into a queue. And the main logic just needs to fetch messages from the queue. Of course, the queue is designed with thread protection by using *synchronized* keyword in Java. When a host wants to send message to another host, it needs to instantiate a "message sender", providing the destination IP and port, and then send messages through it.

SYSTEM TESTING DESIGN

Analysis of Expected Performance

We assume all the testing machines of the same CPU, and network usage. Thus, their performance on the same data size should be the same. It turns that it takes at least 16 seconds for each machine to finish one cycle with 100,000,000 cells. For our project, doubling the number of clients is expected to double the overall performance. Our experiments are performed in SFU CSIL Lab. At least 64 clients are required to observe the performance growing trend. We expected that the performance with 64 clients will be at least 64 times better than a single client. The following figure depicts the expected performance test result.

The horizontal lines indicate the time it takes the system to finish one cycle. When all clients are evenly cut into halves, the overall performance will become twice as before. After 64 clients added, the system is expected to calculate 100,000,000 cells in $16/64 = 0.25$ seconds. In addition, each performance improvement requires twice number of clients than before. Furthermore, the trends will continue to grow until the system switches from CPU-bound to network-bound.

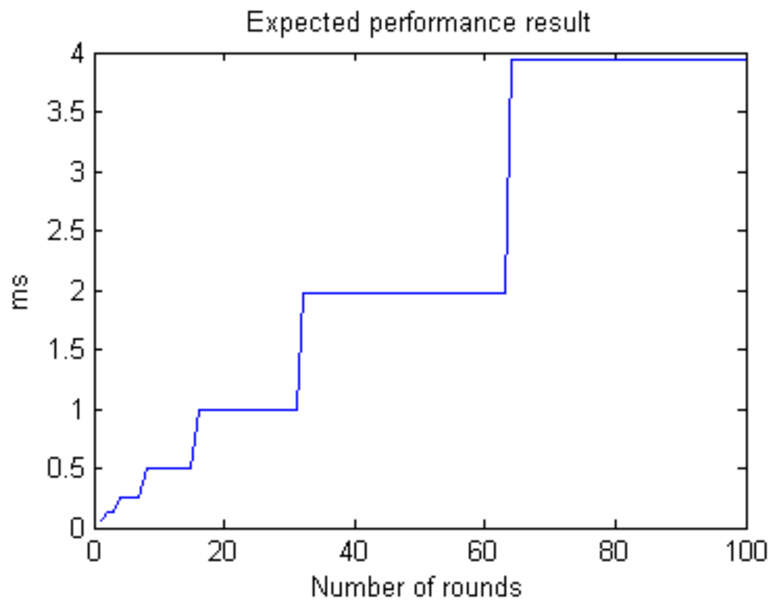


Fig. 11 Expected cycles/sec performance result

Design of Performance and Reliability Testing

We are testing to ensure that the system will perform and be reliable:

- Under high user load
- With a large dataset
- Over an extended period of time
- As the load / dataset increases

Test Objectives

- With the increasing number of clients, observe the updates (cycles) per second.
- The longest computing time in each round
- The average computing time of each client in each round
- The network flow including sending messages and receiving messages

Test Strategy

Our test is designed to use SSH to connect all Linux Computers in CSIL. One teammate ran the server side in one machine, and each of the other teammates connected 30 ~ 40 machines to run as clients by using client JAR. Thus, we successfully added more than 100 clients to the system. We designed to add a new client per 20 cycles, called a new round. Cycle is defined as a transition that all cells in the board finish one update. Incoming clients are added to a pending list. The formula below is to calculate updates per second:

$$\text{Cycles/second} = \frac{20 * 1000}{\text{end Time} - \text{start Time}}$$

During each round, the time required depends on weakest-link principle (Cannikin Law). In other words, the overall time in this round will depends on the slowest client. We measured the longest computing time in each round. In addition, monitor the sender and receiver, and record the size of each package it send or receive.

$$\text{Total network flow} = \sum \text{sender} + \sum \text{receiver}$$

TEST RESULTS AND ANALYSIS

Computing Time

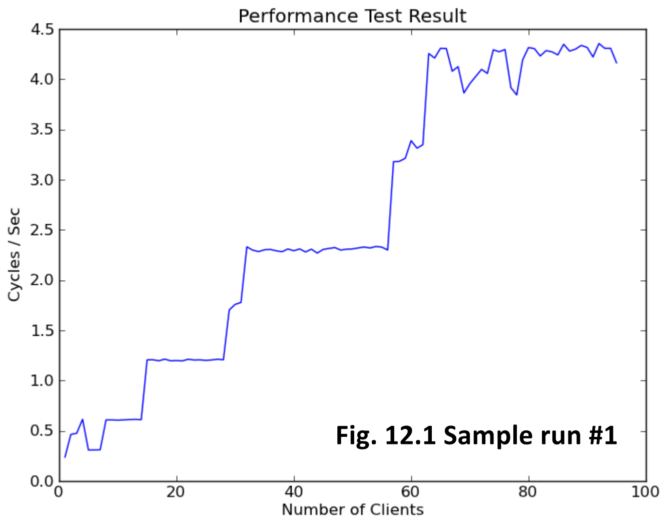


Fig. 12.1 Sample run #1

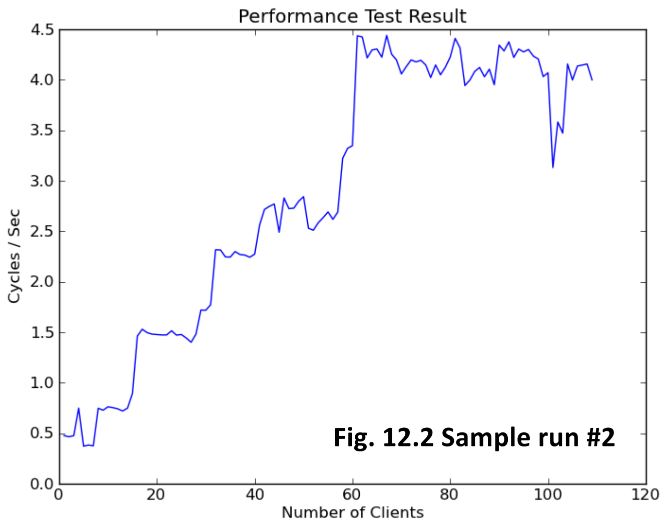


Fig. 12.2 Sample run #2

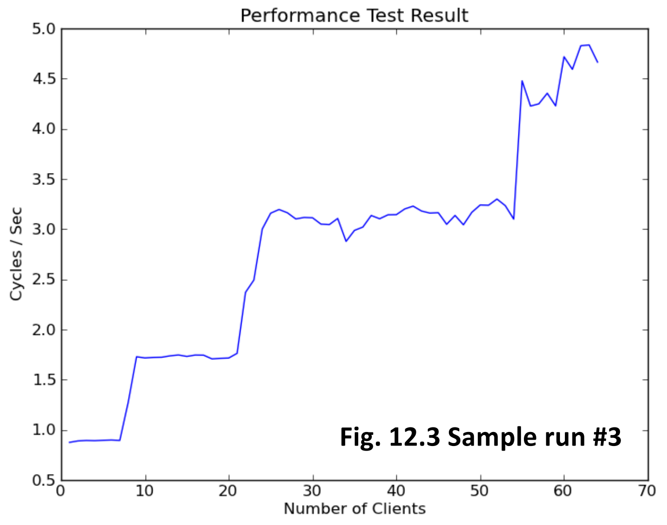


Fig. 12.3 Sample run #3

The result of the first sample run can be seen in Fig.12.1. In this case, 96 clients are added to the system to run a board with 100, 000, 000 cells. The result of this sample run is close to our expectation. As Fig.12.1 indicates, the performance of the whole system increases twice as it is in the last round when the number of clients increases from 2^n to 2^{n+1}. What's more, in the middle of the 2^nth round and the 2^{n+1}th round, the performance of the system keeps as a horizontal line. Finally, the cycles per second metric reached 4.4. This sample run is the most beautiful result among all of our samples, since it is very close to the theoretical result.

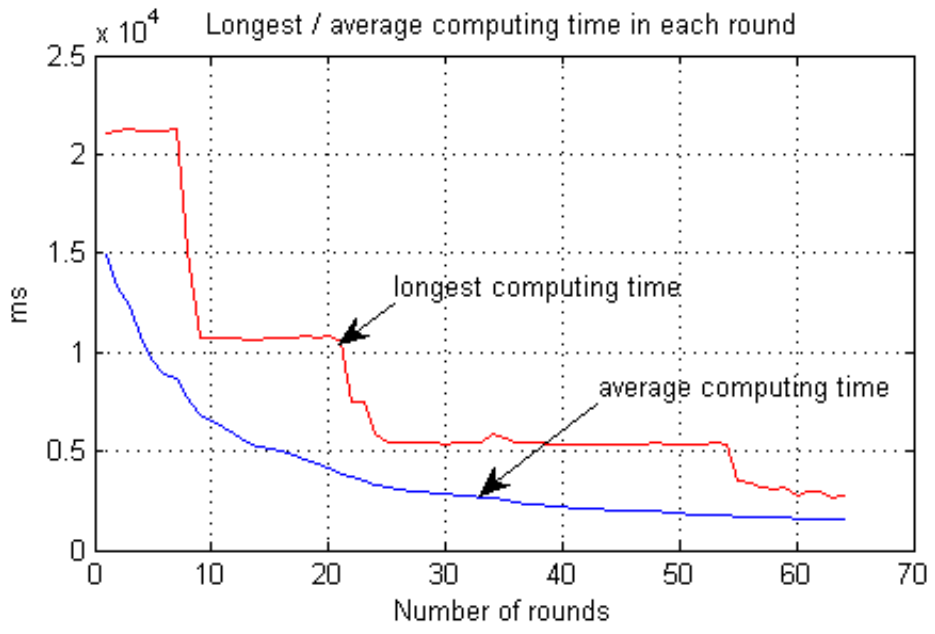


Fig. 13 Longest & average computing time per client result

Fig.12.2 shows the second sample. 110 clients are added, which is the largest number among all of our tests. This line seems to be not as smooth as the first case and it jumps high and drops frequently. It is due to the unstable environment, since a computer in CSIL may be shared by other users and network may be congested.

We want to get more information about the details of the system, so we go on for a third sample run in Fig.12.3. In this case, we compute the average time as well as the longest time of all the clients in each round. According to Cannikin Law[1], the performance should be tightly related to the slowest machine. As we can see in Fig.12.3 and Fig.13, whenever the longest computing time drops to a lower level, the system performance will have an increase. Also, when the slowest machine gets even slower or there is a very slow machine added into the system, the system performance will suffer a decrement.

[1] **Cannikin Law:** How much water a cannikin can contain depends on the shortest board, rather than the tallest one.

For instance, at around Round #23 to #25, the performance of the whole system jumps twice, because the *longest time* drops twice at the same time. Maybe there were two slow machines in the system, whose boards were split into two parts respectively at that time.

Fig. 13 also gives the average computing time in each round, which is a very smooth curve. Although different machines have different computing speed, the average speed in each round is very close to the formula $s = \frac{c}{n}$ where s is the average speed; c is a constant representing the whole computation and n is the number of clients.

Network Bandwidth

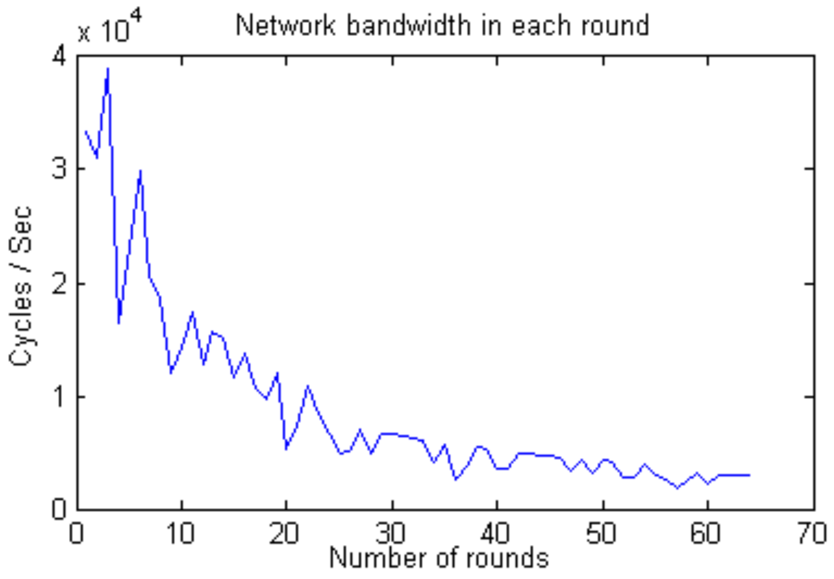


Fig. 14 Average network bandwidth per client result

This result is also from sample run #3. Fig. 14 gives the average network bandwidth that each client occupied in each round. The network bandwidth means the sum of data (measured by bytes) sent and received by the client. As the number of clients increases, each client will send and receive smaller size of messages. At the beginning of the test, the network bandwidth jumps and drops dramatically; it is probably because we add clients at a period of time and this generates more split messages and neighbor updating messages to send and receive. When the adding events were completed, it shows a steady decrement.

CONCLUSIONS

With our one semester's effort, almost all of our goals are achieved. The final system can support dynamically adding and deleting computing nodes. The system has demonstrated its scalability by adding more than 100 clients without any bugs and it is possible to add ten times more clients. The clients can also leave the system without influencing the overall performance. Even if all the clients leave the system, the system is able to continue to run correctly when a new client is added to the system again. It has been tested to support over 100,000,000 cells and 10,000,000 lives. With 100,000,000 cells, the system which is connected with more than 64 clients finishes about 4.5 cycles / sec, better than our expectation.

There is also a large space for us to go further. First, the system cannot handle unexpected error, for example, there is an unexpected crash in a client. We have considered adding backup and rollback to solve this problem and make the system robust even if there is such an unexpected event. But finally we decided to pay more attention to the performance testing due to the time limitation. In addition to that, in our current work, only the server can view the board. It may be more interesting to make clients display a certain part of the board. Lastly, we would like to extend the rule of the life game. It is possible to define some other interesting rules describing how the cells survive or die. Finally, our system can be regarded as a foundation system, which can be extended to simulate practical problems in real life. All of these points provide some good reasons to further improve this distributed system.

REFERENCES

1. Martin Gardner, *The fantastic combinations of John Conway's new solitaire game "life"*, Scientific American 223 (October 1970): 120-123.
2. *Conway's Game of Life*, Wikipedia, http://en.wikipedia.org/wiki/Conway's_Game_of_Life
3. *Typical Uses of Cellular Automata,* http://www.mjyonline.com/CellularAutomataUses.htm
4. Gosper, Bill. *Exploiting Regularities in Large Cellular Spaces.* Physica D. Nonlinear Phenomena (Elsevier) 10 (1-2): 75–80.
5. T. Toffoli and N. Margolus, *Cellular Automata Machines A New Environment for Modeling,* The MIT Press (1986).
6. M.G. Norman, J.R. Henderson, G. Main, D. J. Wallace, *The Use of the CAPE Environment in the Simulation of Rock Fracturing, Concurrency: Practice and Experience* 3, pp. 687 (1991).
7. Domenico Talia, *Cellular Automata + Parallel Computing = Computational Simulation*, ISI-CNR

Appendix I

Message List

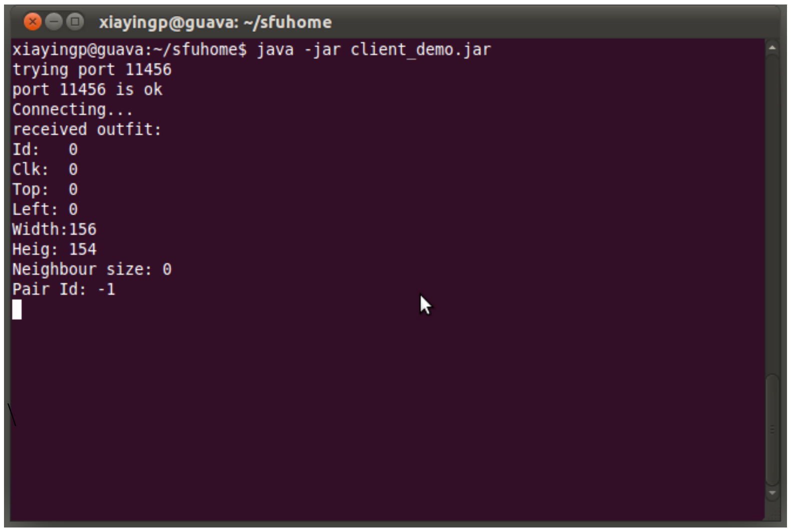
Table 1. System message list

	Messages	Direction	Payload	Description
R1	Next Clock	S→C	new clock number	Inform all the clients to start new computation round
R2	Border	C→C	border status vector	Send border information to clients that is in charge of adjacent area
R3	Report	C→S	whether is leaving, coordinates, status bit map	Report finished computation, and return the computation result to the server in interactive mode
R4	Confirm	A→A	none	Confirm received last message
R5	New Outfit	C→C	connection information, outfit information	Send all the computing information to a new added client
R6	Neighbor Update	C→C	connection information, relative position	Inform all the neighbors about the position change
J1	Request Join	C→S	connection information	Send request to server for joining the computation system
J2	Split	S→C	connection information, new client id, split direction	Inform the right client to split its area, construct the new outfit and send to the new client
L1	Last Merge	S→C	connection information	Inform the last two client to merge into one, the idle one waits for new command
L2	Outfit Merge	C→C	connection information, outfit	Send outfit to pair client for merging
L3	Leave Permit	S→C	connection information	Send outfit to the specified client

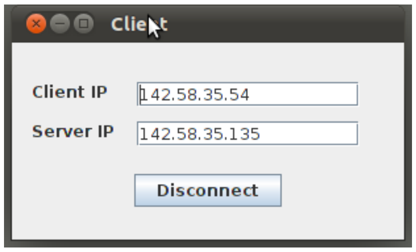
Appendix II

Demonstration of Example System Run

Client Computation



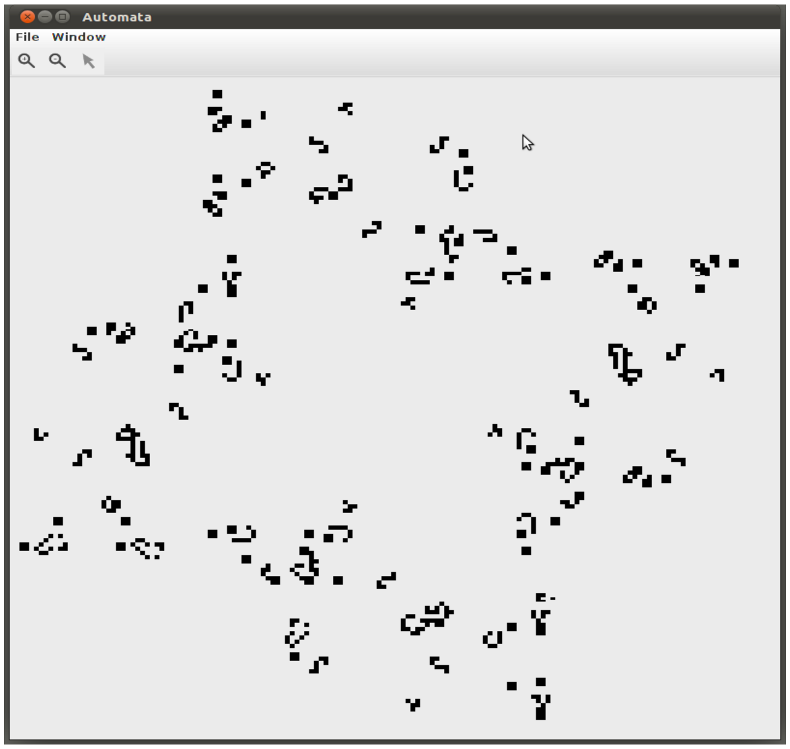
```
xiayingp@guava:~/sfuhome$ java -jar client_demo.jar
trying port 11456
port 11456 is ok
Connecting...
received outfit:
Id:   0
Clk:  0
Top:  0
Left: 0
Width:156
Heig: 154
Neighbour size: 0
Pair Id: -1
```

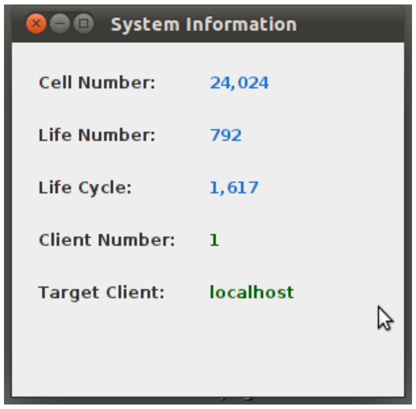


Client IP 142.58.35.54
Server IP 142.58.35.135

Fig. 15 Client running demo

Server Displaying the world of cells and server system information





Cell Number: 24,024
Life Number: 792
Life Cycle: 1,617
Client Number: 1
Target Client: localhost

Fig. 16 Server running demo