Handling Failures: Supercomputers playing Telephone

New Mexico
Supercomputing Challenge
Final Report
April 1, 2012

Team Number 16
Aspen Elementary

Team Members:
Alex Ionkov
Andrei Popa-Simil

Sponsor:
Kathryn Thomas

Mentor:
Latchesar Ionkov

Table of Contents

Executive Summary

There are many ways for supercomputers to break. One example is when the links between the supercomputer nodes break in a way that allows the nodes to communicate, but corrupts the info packets that are sent over them. This is what we will study with our model and experiment. A supercomputer is a machine made out of nodes and links. Nodes do the calculations and communicate with other nodes over the links by sending info packets. Info packets (also known as infos or messages) are small packets of data The problem that we are trying to work on is that the information send over the links sometimes gets corrupted. We created a model of a supercomputer and studied how the number of links between the nodes, and the number of broken links affects the ratio of corrupted info packets. Our experiments have shown that: as expected the less broken links the corruption rate is lower, the more number of links the less corruption rate. We did not get to adding the error-checking and recovery algorithms that would make the info stay healthy but made the environment more realistic. We are planning to work on adding that next year.

Introduction

Supercomputers are the fast calculating machines that are used to solve very hard problems. So what happens when one gets broken from the inside by having it's info packets corrupted? This is what we will model and experiment, if there is a way to reduce the corruption rate.

A supercomputer is a computer at the frontline of current processing capacity, particularly speed of calculation. A supercomputer is a machine made out of nodes and links between them. Nodes are small computers that do the calculations and send info packets to other nodes over the links. Info packets (also known as infos or messages) are small packets of data.

We based our project on the game telephone (but making some changes). In the game telephone (or chinese whispers) there as many players as possible line up such that they can whisper to their immediate neighbors but not hear any players further away. A phrase will be told by the judges and the first player whispers it as quietly as possible to his or her neighbor. The neighbor then passes on the message to the next player to the best of his or her ability. The passing continues until it reaches the player at the end of the line, who says to the judges the message he or she received. The game has no winner but the entertainment comes from comparing the original and final messages. Intermediate messages can be compared too. As well as providing amusement, the game can have educational value. It shows how easily information can become corrupted by indirect communication. The game has been used in schools to simulate the spread of gossip and its supposed harmful effects. It can also be used to teach young children to moderate the volume of their voice, and how to listen attentively; in this case, a game is a success if the message is transmitted accurately with each child whispering. It can also be used for older or adult learners of a foreign language, where the challenge of speaking comprehensibly, and understanding, is more difficult because of the low volume.

Another use is to model data transmission in supercomputers. Some supercomputers function this way. Each node passes the message to it's neighbor node and goes on like that until the message or "info" reaches the final or destination node.

Problem Definition

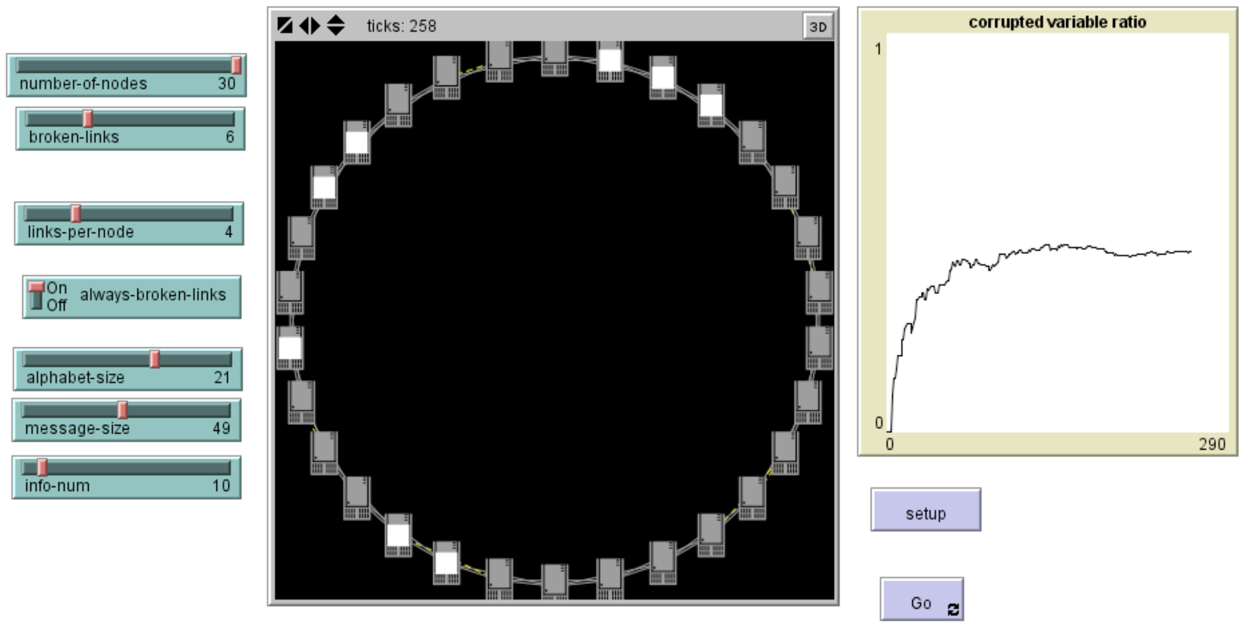
The problem that we are trying to work on is that the information sent by the nodes in a supercomputer sometimes gets corrupted. The links on which the info packet went on were faulty. This is important because supercomputers are used for many things, for example: quantum physics, weather forecasting, climate research, oil and gas exploration, and molecular modeling. Incorrect data can confuse the scientists and make their work harder. We are trying to create a model a supercomputer so we can study info packet corruption. By running experiments with different input variable values, we can try to find solutions for the problem. We will experiment until we have found inputs that will reduce the corruption ratio as far down as we can make it.

Solution

We created a model of a supercomputer in NetLogo. Our model consists of nodes and links between them. We also have infos, which represent the data packets that are send from the nodes to other nodes.

In our model, each node is connected to some (but not all) of the other nodes. If an info is sent to a node that is not connected to the sending node, the info needs to pass over multiple nodes and links.

This is a screenshot of our NetLogo model:


number-of-nodes 30
broken-links 6
links-per-node 4
On Off always-broken-links
alphabet-size 21
message-size 49
info-num 10
ticks: 258

Breeds and Links

Nodes

Our model has a breed called nodes. The nodes represent the supercomputer's nodes. In our model, they are drawn in a circle.

Connections

Connections link two nodes and represent the network connections in the supercomputers. In our model we used NetLogo's unidirectional links. The connections can be either broken, or healthy. If an info passes through a broken connection, it gets corrupted and its content is changed.

Infos

The info breed represents the messages that are passed from one node to another. The infos have many breed variables. They are described in the table below.

Name	Description
start	node that sends the info
finish	node that should receive the info
data	content of the info
location	current location of the info
original-data	original content of the info

We can find out if the info was corrupted by comparing the data variable to the original-data variable.

Input variables

Our model has 7 input variables. We can use them to run different experiments and study the corruption problem.

Number-of-nodes

We can use this input variable to change the size of the supercomputer. We can model supercomputers with 3 nodes up to 30 nodes.

Broken-links

We can use this input variable to change the number of broken links.

Links-per-node

This input variable specifies the number of links connecting each node to other nodes. It can be a value from 2 to 10.

Always-broken-links

In real supercomputers the links can be flaky and not broken forever. If this switch is off, a link is broken for some time, then gets healthy and another random link is broken.

Alphabet-size

This input variable defines how may different values can each character of the content have. If the number is 26, the content of the info can be a random word of letters. If the number is the smallest value 2, the content is something like `bbbaba`.

Message-size

This input variable defines the size of the info content. If the message size is 4, the content is a random four character word.

Info-num

This input variable specifies how many infos are currently being send from one node to another.

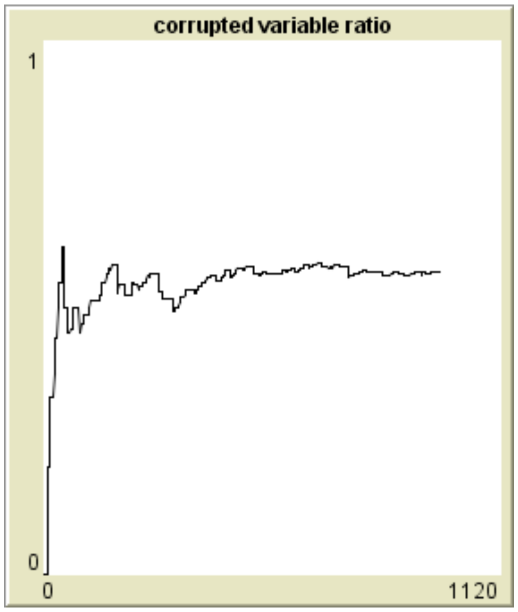
Our program creates some infos with random start and finish nodes, and content values. Each step of the simulation tries to move the infos closer to the finish node. If there is more than one connection, we pick the one that links to a the node that is closest to the finish. If the connection is broken, the program changes one random character from the message.

Results

Figures 1, 2, 3, and 4 show the experiments that we did. The common parameters for all experiments were:

- number of nodes: 30
- alphabet size: 21
- message size: 21
- number of infos: 3

Figure 1: broken links: 3 links-per-node: 2	Figure 2: broken links: 6 links-per-node; 2



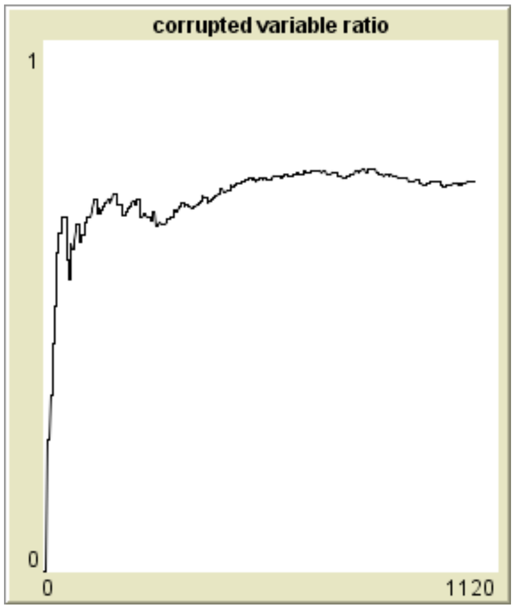
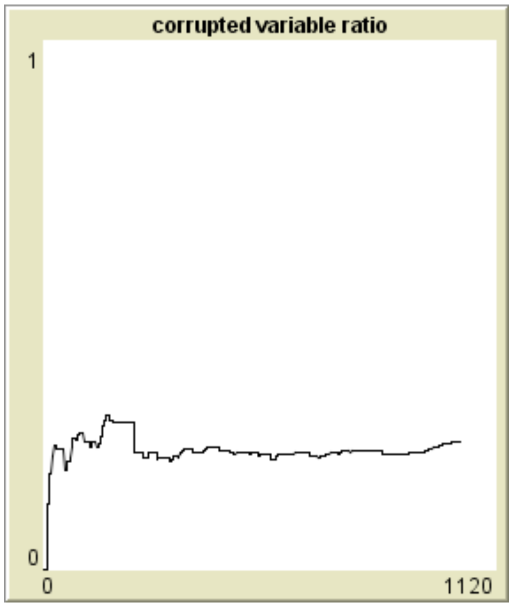
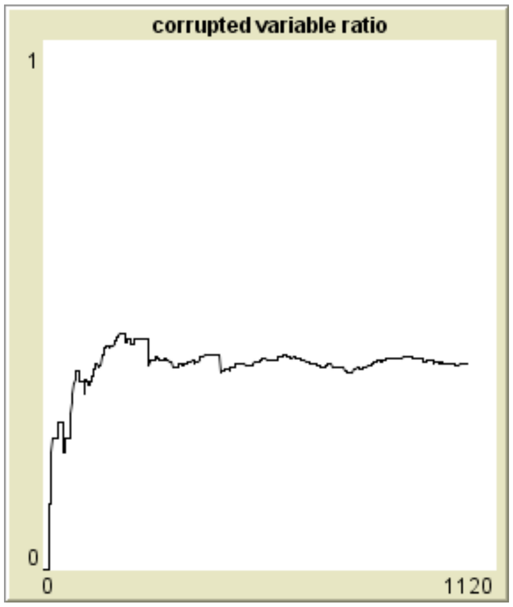


Figure 3: broken links: 3 links-per-node: 4	Figure 4: broken links: 6 links-per-node: 4





As expected the less number of broken connections we have, the less of the infos will get corrupted. If there are more connections between the nodes, the corruption rate is lower. We experimented with different alphabet sizes (results not included) and found that the smaller the alphabet size is, the lower the corruption rate is. The number of nodes affected the corruption rate by giving the infos less places to go.

Conclusion

Our simulation showed that the info/infos can be corrupted very fast resulting to losing data. We have made the right experimenting environment to study the supercomputer data corruption. If we extend this project, we will add error-checking and recovery algorithms. Another feature we would like to add is to detect which links are faulty and not send data over them.

References

1. Cyclic redundancy check, http://en.wikipedia.org/wiki/Cyclic_redundancy_check

2. Error detection and correction, http://en.wikipedia.org/wiki/Error_Control_Coding

3. Supercomputer, http://en.wikipedia.org/wiki/Supercomputer

4. Computer network, http://en.wikipedia.org/wiki/Computer_network

5. Telephone, http://en.wikipedia.org/wiki/Chinese_whispers

6. Netlogo Dictionary, http://ccl.northwestern.edu/netlogo/docs/dictionary.html

Acknowledgments

We would like to thank to Latchesar Ionkov for being our mentor and Supercomputing Challenge program for helping us to learn computer programming.

Appendix 1: Source Code

```
breed [nodes node]
breed [infos info]
undirected-link-breed [ connections connection]

connections-own [ broken ]
infos-own [location start finish data original-data]

globals [
  alphabet
  info-template
  broken-info-num
  total-info-num
  info-ratio
  current-broken-links
]

to setup
  ;; (for this model to work with NetLogo's new plotting features,
  ;; __clear-all-and-reset-ticks should be replaced with clear-all at
  ;; the beginning of your setup procedure and reset-ticks at the end
  ;; of the procedure.)
  __clear-all-and-reset-ticks
  set-default-shape nodes "computer server"
  create-nodes number-of-nodes
 layout-circle sort turtles (world-width / 2 - 1  )
  ask turtles [ set size 4 ]
  link-nodes
  set alphabet substring
 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
0123456789-,.<>:[]{}!@#$%^&*()+=" 0 alphabet-size
  ;tick
  set info-template substring
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
0123456789-,.<>:[]{}!@#$%^&*()+=
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
0123456789-,.<>:[]{}!@#$%^&*()+=" 0 message-size
end

to link-nodes
  ask nodes [
    let id who + 1
    repeat links-per-node / 2 [

      if id > (number-of-nodes - 1) [
        set id 0
      ]
      create-connection-with node id
      set id id + 1
    ]
  ]
  setup-broken-links
end

to setup-broken-links
  repeat broken-links [
    break-link
  ]
end

to break-link
  let dostop false
  while [not dostop] [
    ask one-of connections [
      if broken = 0 [
        ifelse always-broken-links [
          set broken 1000000000000000
        ]
        [
          set broken 1 + random 60
        ]
        set shape "default1"
        set color yellow
        set dostop true
      ]
    ]
  ]
  set current-broken-links current-broken-links + 1
end

to-report best-link [ finish-id ]
    let new-location 0
    let dist 10000000
    let id who
    ask connection-neighbors [
      let d abs(finish-id - who)
      if d < dist [
        set new-location self
        set dist d
      ]
    ]
    report ( list new-location link-with new-location )
end

to info-step
    if location = finish [
      if data != original-data [
        set broken-info-num broken-info-num + 1
        set info-ratio broken-info-num / total-info-num
      ]
      die
      stop
    ]
    let finish-id [who] of finish
    let l [ best-link finish-id ] of location
    let next-node first l
    let next-link last l
    set location next-node
    let new-data data
    ask next-link [
      if broken > 0 [
        let pos random (length new-data)
        let lpos random length alphabet
        set new-data replace-item pos new-data (item lpos alphabet)
      ]
    ]
    set data new-data
end

to check-broken-links
  ask connections [
    if broken > 0 [
      set broken broken - 1
      if broken <= 0 [
        set broken 0
        set color white
        set shape "default"
        set current-broken-links current-broken-links - 1
      ]
    ]
  ]
  repeat broken-links - current-broken-links [

    break-link
  ]
end

to go
  ask infos [
    info-step
  ]
  check-broken-links
  if count infos < info-num [
    repeat info-num - count infos [
      new-info one-of nodes one-of nodes random-data
    ]
  ]
  tick
end

to-report random-data
  let di info-template
  let pos 0
  while [pos < message-size] [
    let lpos random length alphabet
    set di replace-item pos di (item lpos alphabet)
    set pos pos + 1
  ]

  report di
end

to new-info [s en d]
  create-infos 1 [
    set start s
    set finish en
    set data d
    set original-data d
    set color white
    set size 2
    set shape "square"
    set location s
    move-to location
    set total-info-num total-info-num + 1
  ]
end
```

Appendix 2: NetLogo

NetLogo is a programming language written by Uri Wilensky in 1999. He wrote it in Scala with some Java in it. He based Netlogo on the programming language Logo, and thus having Logo in it's name. Net in NetLogo comes from Netlogo's ability to make networks. It was made to be "low threshold and no ceiling," that is to enable easy entry by novices and yet meet the needs of high powered users. For more information go to http://en.wikipedia.org/wiki/NetLogo or http://ccl.northwestern.edu/netlogo/.