Cha Li
11.28.2011
CS2510
Project 3 Proposal – Crowd Computing: Distributed Sudoku

Introduction

I'm not sure if the term “Crowd Computing” is already used to describe something else, but I thought it was a good (it reeks of buzzwordiness) term to use for describing a paradigm that is the opposite of Cloud Computing. The idea is that companies, governments, and other organizations keep track of large amounts of data but instead of processing it on dedicated machines the data is distributed over many mass-consumer grade computing devices. Although not as powerful, there is a large pool of unused processing potential that can be tapped into for doing productive work. This idea isn't new however, a project called “Folding@home” takes advantage of spare processing on Personal Computers (PC) to carry out protein calculations. The PC simply runs a screensaver that does these calculations. More info about this can be found here: http://en.wikipedia.org/wiki/Folding@home

Motivation

The motivation behind my proposal is based on the prevalence of smartphone ownership. Before the advent of the *mass-consumer* smartphone, PC's were the most popular personal computing platform and it's popularity operated on the per family level (i.e. 1 or 2 PC's per family). However, fast-forward to present day and the smartphone has become the most popular personal computing platform. The scale at which smartphone ownership operates on is the *per-person* level, this quantity easily overshadows PC ownership. In addition to it's numerical dominance, smartphones offer other advantages over PC's that can be harnessed for a distributed computer system. Firstly, smartphone's are “always-on” as a result of the relationship people have developed with the technology. With PC's, a user will usually logout/lock/shutdown the system when they no longer need it which essentially prevents the PC from sharing its spare cycles at a time when all the PC has is spare cycles. On the other hand, smartphones are always on. Even when the owner goes to sleep for 8+ hours, the smartphone remains “live” but sits idle for the same time period (in most cases it is plugged into a constant power source too). Secondly, smartphones are “always-in” meaning they are always (or close to always) connected to the internet (wirelessly (wifi, 3G, 4G ,etc)), which can't be said about PC's which require a wired connection or access to a wifi network (which aren't as prevalent as nG networks). Thirdly, smartphones contain, relatively, a lot of computing power in a small space. I think most modern smartphone have dual-core CPU's running at 1Ghz with 1GB of memory and 16+ GB of “disk space”. All this processing power allows users to watch high-definition video and play games, but I'd say the majority of time spent on smartphones is not used to do those things (at least 8 hours are spent doing nothing while the owner sleeps). It's also safe to assume that the computing power of smartphones will increase. So what does this all mean for distributive computing? It means that in a relatively small geographical space, there is a large amount (100's of cores in a small skyscraper, perhaps 1000's in a large skyscraper) of extra processing power that is always on and always connected just waiting to be tapped. I intend to build a small prototype system (< 10 phones) that illustrates the potential of this available processing power by having it solve a simple game of Sudoku.

Distributed Sudoku

I like the idea of crowd-computing and the idea of contributing spare processing power to a greater cause, but I needed to think of a practical application of it that could illustrate its problem solving potential. It couldn't be too large of a problem since I'm constrained to two weeks but I also didn't want it to be so small that the project became trivial. Since my interests are in AI, I figured Sudoku would be a good application, with the added twist that it's solved with a distributive system – Distributive Sudoku.

Hardware Requirements

I wanted to avoid simulating nodes for my final project, since having actual physical nodes is much cooler. I currently possess two Android phones, 1 personal and 1 for development, which I can use as nodes for this project but I will be attempting to get more (ideally 9).

In addition to the phones, I will use my laptop as the server that generates Sudoku boards and use my router to provide the network. I'll probably have to bring all these components in for the demo.

Software Requirements

This project requires one Android application (henceforth called “the application”) which wraps some old Sudoku code I have (which needs to be converted to Java from C++) in a really simple GUI and provides the basis for a distributed system (see below). In addition, some other code has to developed, most likely in Java, to act as the server on my laptop.

Architecture
One central server (my laptop) used for generating Sudoku boards and general management of the system. The server will be mostly hands-off when it comes to the actual solving aspect.

N Android devices, nodes, which will handle solving the Sudoku board together. Each node will be given a sector of the Sudoku board and tasked with solving it. A sector is defined as one of the nine “major” squares that consists of nine individual “minor” squares. However, this distribution isn't set in stone yet, I am still considering other options such as dividing the work by rows, columns, or even numbers. Essentially, each node will be given a task that is independent of another nodes task. Where independent is defined as nodes not solving the same physical areas of the Sudoku board.

Communication
Solution related communication will be handled entirely by the nodes, the server will have little(no) involvement with the problem solving process.

A node will solve as much as it can based on what it currently knows of the state of the board, each valid number that the node fills into its sector will be broadcasted to all other nodes in the system. When a node has exhausted all valid numbers, it waits for information from other nodes before moving forward with its own sector.

I am removing the ability to guess from the system which means numbers can only be placed in their final valid positions. This simplifies the problem a little by reducing the importance of mutual exclusion when it comes to placing numbers. Given this simplification and the rules of Sudoku, N nodes will never be placing N different numbers in the same position or placing the same number in N different positions. It's possible I'll still need some ME when it comes to executing forward-checking and arc-consistency on the board to prevent changes from happening during the check.

Power Management
One of additional benefits of developing an Android application is the ability to control its state (see http://developer.android.com/guide/topics/fundamentals/activities.html#Lifecycle). The states of interest are “running” and “paused (idle)”. The Android documentation describes the paused state as:

> This [state] is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.

For the power management aspect of this project, I will be taking advantage of this. The node can put itself in an idle state if it can't make any more progress in its given sector (since there's nothing to process). The node can also be put into the idle state by the central server. For example, if the server is told to operate at 50% capacity , it can tell N/2 nodes to start idling (after collecting their state info). Although there is no way for me to calculate exactly how much power is being saved, it can be inferred (from the above description) the application is using less power since it no longer uses the CPU. I would need lower level access to actually throttle the CPU.

Real-Time Requirements
This is an interesting requirement, since Sudoku inherently doesn't have any real-time requirements. However, to make things interesting I will implement real-time constraints into the system by dividing the system into two modes: solving and checking. The available time will be split into alternating slices each devoted to solving the board and checking the board. For example, solving will be divided into five-second slices and checking will be divided into ten-second slices. I'm currently not sure if these will be hard or soft deadlines.

Another bonus that comes out of this is that it removes the need for mutual-exclusion since solving and checking are now two separate activities. I'll probably go with this system.

Fault Tolerance
There are a couple assumptions I am basing my system on: The server never* dies and the communication network is 100% reliable. That being said, nodes can and will die. In the context of this project a node dying means the smartphone meets one , or more, of the following conditions: it's off, it's battery is dead, it's not connected to a network, and/or the application is not running, The server simply recognizes all these conditions as disconnected. There are two cases when a node can become disconnected each one with a different detection method, but same recovery method.

The first case is the smartphone meets one of the above conditions. This is considered a client-initiated death, and its detection isn't immediate but is pretty simple. Since each sector of the Sudoku board is dependent on other parts of the board making progress, a death can be detected by the server (and other nodes) if all surviving nodes stop making progress for some duration of time (to be defined later).

The second case is server-initiated deaths. This happens when the server is told to operate at a capacity < 100% forcing it to put some nodes into the idle state. Obviously the server will detect these deaths immediately since the server initiated them.

In both cases, recovery begins with the server telling all surviving nodes to stop working on the Sudoku board and asking them for their knowledge of the current state of the Sudoku board. Once the server collects all the boards it determines which board contains the most progress (most correctly placed numbers) and when appropriate it merges boards to create a board that represents total progress of the system. When the recovered state is created, the server broadcasts it to all the nodes and redistributes sector assignments among the surviving nodes.

It should also be mentioned that the recovery progress can be done distributively, but for now I am going with a centralized approach unless I have some extra time near the end to work on a distributive solution.

*Theoretically the nodes can still go about solving the Sudoku board if the server dies as long as the system remains static (nothing dies).

Load Balancing

The recovery process above is basically the load-balancing arm of this system. If a node dies or joins mid-solving the server just recalculates the state and redistributes the sectors. The current scope of this project only involves the nine sectors of a Sudoku board so they can only be evenly distributed across a maximum of nine smartphones. If the time spent redistributing the task, due to joining nodes, overtakes the time spent solving the task, the server can ignore giving new nodes tasks. The server must always redistribute the board if a node dies. In the former case, the system still contains all sectors so progress can still be made without giving the new node a task, however, in the latter case the sectors must be redistributed since a dead node means the system is missing sectors.