Misc...

Alex S.*

1 Introduction

These notes are about a whole variety of topics...

2 Implementing a Network

There are a few important things you need to do when setting up a network. For one, you need to understand the need for the network, understand what capacity/response time it needs to have, etc. You also need to get the right equipment for the job (consumer grade routers aren't always appropriate in a corporate setting, etc.). You also need to figure out the costs—networking equipment can get *really* expensive.

Once you pick technology, topology, approach, etc., you actually need to implement it—correctly. You need to know how to properly work with the hardware/software, configurations, etc. There are far too many companies with unsecured WiFi simply because someone didn't take the time to setup things properly (or how about open proxies!).

Another important aspect is to plan for down-time, do routine maintenance (and have some backup plan if *everything* decides to not work all of a sudden—and it does happen).

Once the network is up and running, it is extremely important to monitor it. To make sure that things are running smoothly all the time. Possibly ping key points from time to time (and e-mail/telephone appropriate people if there is no response).

Ensure there is someone to contact in case of a failure/problem. Ensure there is a place that assigns logins, permissions, passwords, etc., and that everyone knows about how/what/where, etc. There should also be a place where users get training on the basics (like not leaving their passwords taped to their monitor—and not click on everything that comes in their e-mail).

It's also important to monitor the network for all sorts of interesting reasons. Some tools that you may find interesting are: Nessus[1] and NeWT[2]. This leads to another issue: sign up for various `bugtraq` mailing lists, and actually read what's going on. Basically stay on your

[1] `http://www.nessus.org/`

[2] `http://www.tenablesecurity.com/`

toes when it comes to your network—don't assume that if things are working today, they'll work tomorrow.

For various reasons you might need to ensure some quality of service for various applications. So that no single user can flood the network, etc., and possibly disable some services. You can do that on Linux with iproute2[3].

Firewalls: The *only* things the outside world should see of your network is precisely the services that *need* to be seen. Many worms in the past few years have spread via something like Microsoft SQL Server—which has no business being accessible over the Internet to begin with. In fact, beware of anything other than SSH and HTTP traffic that goes in and out of your network—all other ports should be closed. Another good strategy is to run every service under its own user account—that way even if say Apache gets attacked, it will just crash (or compromise) that account—and not the whole system.

3 Super Servers

There is a concept called the 'Super Server', and it mostly fits in the Unix/Linux world. The idea is you have one program, that listens on many many ports, and when someone opens a connection on any port, that software determines which program to run to handle connections on that port. This has several benefits: you can control most of your 'common' network services from a single location (and you don't need to 'restart' each service when you upgrade it—since every request user restarts the service), and best of all, the services aren't 'always' running and consuming resources—they only get instantiated when someone's using them. You can view it as the low level version of CGI (the port determines which program gets executed to handle IO). Two examples of super servers are: `inetd`, and `xinetd`. They're usually used for things like the 'finger', 'talk', etc.

4 Load Balancing

In many situations you'd like to support a large number of users, and it's impractical to have a single computer to do the job. For example, if you want to have a website that can handle a few hundred thousand users, etc.[4]

Most load balancing solutions work by forwarding the user somewhere else. This can take several forms: forwarding the user on every request, or forward the user on the first quest, afterwhich the user just talks to a single machine.

A similar idea applies to databases, but those are a bit harder to distribute. Since querying the database usually accounts for majority of the database load, it is possible to setup several 'read' databases (like 15 or so), and setup only one 'wright' database. When

[3] `http://developer.osdl.org/dev/iproute2/`

[4] Most web-hosts now a days can handle a relatively heavy load—but most will still go down due to a food of connections from such things as the Slashdot Effect: where thousands of nerds click on a link at the same time.

someone reads, their query goes to a random server out of the available bunch, when they need to write (update) something, their request goes to a specific server (the update is then propagated to the read servers—usually after a certain time period). This type of a setup can handle enormous database loads (usually using many relatively primitive servers—like MySQL).

Just as with database, you can have different application servers (or web-servers) for different tasks. For example, many high-load web-servers separate their dynamic content and their media content to different servers—so for example, the text may come from one server, while images may come from another (or from one of many available servers).

In some cases, entire user profiles are stored on a certain cluster of the whole system. For example, when you login into Yahoo!, you are redirected to 'your' server (or the server that's responsible for your user account). You interact with that server, can update your user data, etc., and all this time, the server itself doesn't really deal with millions of records—it just has a subset of records for which it is responsible. This type of setup allows handling unlimited number of users—because you can always just add more machines to handle larger loads.

5 Video Game Communication

Note that most of this applies to most online multilayer games—I'm sure there are exceptions.

There is usually a common 'game' server. It maintains the state of the game, all players, game objects, etc. It runs at its own time (usually all actions happen on the server). A client (or many) connects to the server, retrieves some initial state, and loads the appropriate map, then 'joins' the game (the server adds that player to the list of playing clients).

The server, periodically, takes the game state, and sends it to all clients, usually using UDP (and usually encoding the whole game state in a single packet). The clients, when they're moving, send those events to the server—which then validates whether those moves were valid—changes its state. All other clients know about this change of state on their next update from the server.

There are some optimizations/improvements to this. Sometimes each packet only encodes part of the state (the most current actions)—but the whole state is sent every once in a while still (to ensure nobody gets out of sync by too much). There are also all sorts of 'prediction' models that each client can run—for example, a player shoots a rocket (notifies the server), that flies and kills the opponent. The player's screen shows the rocket and, by prediction, it's consequences.

The server, gets the message, does the calculation (independently) and either: comes to the same conclusion as the client (the rocket hits the other player), or the other player moved away and the rocket misses. The server notifies everyone of its state—and the original client may see a 'jump' in the game—where one moment they saw the rocket hit someone (or heading right at someone) and then miss. Now a days, this only happens on a really slow connection and only in situations where the network dies for a few seconds. Most games are capable of sending about 40-90 packets per second, and receiving them at a much higher rate.

6 Various Open Problems

There are some interesting problems that have yet to be properly solved.

These include Network Testing. There is no systematic way to 'test' a network. The best that anyone can do is run a few stress tests, and then declare the network to be 'working'. Unfortunately there are various time dependent factors that make this approach not very reliable. You might stress test the network, and say "yes, it works", only to find that everything stops working because some primary server stopped responding the next day when its log file filled the whole drive (ie: things break in places where you least expect them—so testing for them is next to impossible). There are various stress testing tools available commercially, etc., and for a large installation, you should probably use them (ie: on a "better than nothing" principle).

Another interesting problem involves sensor networks[5] or ad-hock networks. In many applications of networking, you need to have a few network nodes to just come together and form a temporary network—to route packets to a particular destination, etc. As it is, there is no *good* way to have those nodes organize and become a reliable 'network'—think of a bunch of people with WiFi—how to make all those WiFi networks into a bigger network—automatically and on the fly.

There are similar problems relating to selecting frequencies for communication in wireless networks. For example, two wireless devices cannot be using the same frequency if they're near each other (they'll get interference). This is a major problem for mobile phone towers, that need to be located in some relative density to ensure coverage, but cannot use conflicting frequencies if they're within a certain distance of other towers that use the same frequency—and there is only a limited range of frequencies to choose from.

[5]Network of small devices that talk via radio and have some sensor, and limited energy, memory, processing power.