Introduction

This set of articles consists of scribe notes from the course CS6282, "Very Large Scale Distributed Systems", conducted in School of Computing, National University of Singapore. These notes are meant to be dynamic, evolving over time, with frequent changes and updates.

1 Introduction

Scribed By Ooi Wei Tsang, version 0.1, Last Modified 11 Aug 05

1.1 DISTRIBUTED SYSTEMS

A distributed system, simply put, is a system with multiple computers working together to perform a task. As an example, consider "viewing a web page" as the task. To perform this task, the web server, caching proxy, ad server, and the end-user's computer work together to retrieve the web page and its object for display at the end-user's computer.

In this course, we are interested in large scale distributed systems. In other words, systems with *a* large number of computers working together to perform a task. An example of such system is the Internet, where tens of thousands of routers are working together to route packets between hosts. Recent technology advances, including faster CPU, higher network capacity, smaller transistors, has lead to many new services, applications and networks that can be characterized as large scale distributed systems. Understanding how these systems work, and how to design and build them properly, thus has became an important issue.

As an example of large scale distributed system, consider a recent phenomenon used for peer-to-peer file sharing called BitTorrent[1]. BitTorrent allows thousands of hosts to share files. This ability is enabled by the trend of increasing high speed connections to homes and increasing harddisk space to store files.

Another example is wireless sensor networks. In such networks, thousands of tiny sensors can be scattered over an area, working together to achieve a task, such as environmental monitoring. Advances in radio and hardware technology has enabled this new type of network.

1.2 DESIGN REQUIREMENTS

We now examine some general properties of a "good" large scale distributed system. These properties should be the design goals, or design requirements of any such systems, and they guide the performance metrics that we should use to evaluate a system.

We should note that it is often impossible to meet all these requirements at the same time. In most cases, we have to trade-off between these requirements.

Correct The first requirement is, of course, the system must behave correctly. Consider Internet routing – the packets must be routed correctly between the hosts. While this requirement seems obvious, Fischer, Lynch and Paterson showed in 1985 that in the presence of unreliable processes, consensus among the processes cannot be achieved [2]. This result implies that, in theory, absolute correctness cannot be achieved if some processes or the network is unreliable. In such cases, we will have to settle for "practically" correct. If you feel uneasy over this, you can think about TCP – a transport layer protocol that is considered as *reliable*. In theory, however, TCP can never guarantee that the packet delivered will arrive (for instance, the network may fail or connection may timeout due

[1] http://www.bittorrent.com

to long delays). But, TCP is good enough to be called reliable. Another point to note is that, in some cases, we will have to sacrifice correctness in order to improve performance in another metric. For instance, to monitor temperature in a sensor network, we may settle for an approximate average temperature, rather than the absolute average, just so that the scheme is efficient.

Scalable Scalability is another important requirement for building large scale systems. The term scalability, refers to the ability of a system to perform well despite increasing demand. We typically refer to scalability with respect to a certain system parameter and a specific task. For instance, Internet can route packets pretty well even as the number of hosts and routers grows. However, existing Internet does not scale in terms of host naming as the number of hosts increases.

Scalability is often relative and subjective. For instance, using BitTorrent to distribute a software release, is more scalable than distributing a new release through a FTP server. But using a centralized tracker limits the scalability of BitTorrent, and thus is less scalable compared to a solution that uses a distributed tracker.

Robust The term robust, or fault tolerant, refers to the ability for the system to continue working in full or in part despite failures or broken part. The Internet routing structure is somewhat robust, as when one or two routers fail, most of the time packets can find an alternate paths between hosts.

Consider the example of BitTorrent. The BitTorrent protocol is extremely robust against failure of peers. It, however, relies on a centralized *tracker* to track who to download files from. If the tracker fails, a BitTorrent client will not be able to function.

This goal closely relates to the goal of *Availability*, which is the percentage of time a system is able to function.

Efficient Another general design goal is efficiency. We should design a system to be efficient in terms of number of messages exchange, computations required, time taken, battery power consumed etc. The latter is especially of concern in a wireless sensor network.

Simple There is a funny article by Doug Comer about how to insult a computer scientist[2]. A quote from the article, says, "Systems researchers take pride in being able to invent the simplest system that offers a given level of functionality". Simplicity, besides being beautiful and elegant, can lead to easier implementation, less bugs, easier maintenance, and easier analysis.

This design goal is known as the KISS principle ("Keep it Simple and Stupid").

An example of a simple design, is how Internet routing protocol deals with router failure or configuration changes. There is no explicit error handling in the protocol. Instead, a router simply periodically forgets routing table entries, and relies of periodic refresh messages from other routers to update the table. Entries corresponding to failed routers will automatically be removed from the table (such states are known as *soft states*).

Ease of Use A design goal related to simplicity is ease of use. The system must provide friendly interface to end-users and the right programming abstractions to developers. The system must support incremental deployment of software and protocols.

Secure Finally, we should design the system to be secure – robust against unauthorized access, tempering and attack from malicious users. In certain applications, ensuring the users' privacy is important as well.

[2]http://www.cs.purdue.edu/homes/dec/essay.criticize.html

1.3 CHALLENGES

In this section, we discuss why it is challenging to meet the design requirements listed in the previous sections. The characteristics of a large-scale distributed system are as follows.

Numbers The first factor that complicates the design of any large scale distributed system is simply the number of hosts or devices involved. Designing a system that scales to a large number becomes important. The efficiency of the algorithm involved becomes important – for instance, a $O(N^2)$ algorithm might be acceptable in a distributed system with 10 nodes, but in a large system with tens of thousands of node, such algorithm will not scale.

The large number of nodes increases the number of faults that can occur. An event that occurs rarely on a single host, can happen pretty often when the number of hosts is large [3].

Scattered The second complications come from the fact that the hosts and devices might be scattered over a large geographical area. This increases the communication latency among the hosts and increases the unreliability of communications among the hosts.

Note that not all large scale distributed systems are scattered. Large cluster of PCs, such as the one used by Google to answer web search query [1], is an example.

Dynamic Users in a large scale distributed system can join and leave as they want. This activity is sometimes known as *churn*. High churning rate poses a challenge as states might needs to be updated or transfered among the nodes everytime a node leave or join the system.

BitTorrent is an example system that can handle the transient of nodes very well.

Unreliable Nodes in a large scale distributed system can fail. In contrast to churn, where a node might leave gracefully, nodes failure occurs unpredictably. Due to the large numbers of hosts, failure can happen more frequently than one would think.

Untrustable Not all nodes in a distributed system can be trusted.

This characteristic is particularly critical in design of distributed games. In games, players are selfish rather than cooperative. They tend to cheat.

Resource Constraint A system always has a bottleneck that prevents it to scale arbitrarily. Such constraints often force us to design efficient algorithms and techniques.

Note that in any systems research, interesting problems exists only when there is certain level of resource constraints. When resources are too limited, there is nothing we can do. When resources are abundance, we can do anything!

Heterogeneity To complicate matters, not only that nodes have resource constraints, different nodes might have different constraints. In a large scale distributed systems, nodes are more likely to have different amount of CPU power, battery power and network link capacity. Some nodes have more resource constraints than others. Such heterogeneity needs to be considered as well. Often, we need to treat such nodes differently.

1.4 BASIC PRIMITIVES

Many different services and applications exists as a large-scale distributed systems. However, most of the applications can be built on-top of a small number of primitives. In this sections, we attempt to list down some of these primitives.

Lookup/Update Table lookup is a primitive operation in many applications. For instance, DNS performs lookup on host name; File sharing applications perform lookup to find out which peer store a particular file.

All lookup-based applications have a naive, unscalable solution in which the lookup table is stored in a centralized server and every node query that server for information (Incidentally, this is how the original Napster work).

If the table is static, then a simple solution to scalability is replication. Most of the time, however, the lookup table is dyanmic and can be changed through update operations. The design space to consider can be different when the lookup table can be changed thorugh insert or delete operations.

One-to-Many Another primitive is dissemination of information to a subset of nodes in the system. This operation, also known as *multicast*, is extremely useful for group communications. For instance, in a video on demand application, the server needs to stream a video to multiple viewers. In a sensor network, a node might want to send messages to query the temperature of a set of other sensors.

The special case where a message is sent to all other nodes is called *broadcast*.

Many-to-one The reverse of the multicast primitive, where multiple nodes send messages to a single sink, is useful for collecting information. An example where this is used is network management. Another example is sensor network. Continuing the previous example, the temperature readings of the sensors can be sent back to the querying node using this primitive.

Publish/Subscribe This specialized primitive, is a more general version of multicast. In multicast, we want to send identical messages to a set of nodes. In publish/subscribe system, different sets of nodes might be interested ("subscribed") in different information. A set of messages needs to be sent ("published") such that only those who are interested in the messages receives it.

An example is event notification services. IBM, for example, uses such service to send updates to millions of users during the Sydney Olympic in 2000.

Maintaining Global States Besides the communication primitives above, applications might require maintenance of some global states across the nodes in a distributed system. One such example is multi-player games.

Read/Write/Execute Some primitive OS operations might be needed for certain applications. For instance, Grid computing allows code to be executed on a remote machine. Download files from a P2P file sharing system, can be considered as reading from a distributed file system.

1.5 SUMMARY

In this lecture, we discussed the requirements, challenges and primitive operations in large scale, distributed systems. The discussions here provide a framework in which individual technique and system that we will study over the semester can be compared and categorized into.

REFERENCES

1. L. A. Barroso, J. Dean, and U. Hlzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(02):22–28, 2003.

2. M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

3. S. Muir. The seven deadly sins of distributed systems. In *The 1st USENIX Workshop on Real, Large Distributed Systems (WORLDS 2004)*, San Francisco, CA, December 2004.