Distributed Systems Goals & Challenges

Before we start discussing distributed systems architectures it is important to understand why we have been driven to build distributed systems, and the fundamental problems associated with doing so.

Goals: Why Build Distributed Systems

Distributed systems could easily be justified by the simple facts of collaboration and sharing. The world-wide web is an obvious and compelling example of the value that is created when people can easily expose and exchange information. Much of the work we do is done in collaboration with others, and so we need often need to share work products. But there are several other powerful forces driving us towards distributed systems.

Client/Server Usage Model

Long ago, all the equipment I needed to use in connection with my computer was connected to my computer. But this does not always make sense:

- I may only use a high resolution color scanner for a few minutes per month.
- I may make regular use of a high speed color printer, but not enough to justify buying one just for myself.
- I could store my music and videos on my own computer, but if I store them on a home NAS server, the entire family can have access to our combined libraries.
- I could store all my work related files on my own computer, but if I store them on a work-group server, somebody else will manage the back-ups, and ensure that everyone on the project has access to them.

There are many situations where we can get better functionality and save money by using remote/centralized resources rather than requiring all resources to be connected to a client computer.

Reliability and Availability

As we come to depend more and more on our digital computers and storage we require higher reliability and availability from them. Long ago people tried to improve reliability by building systems out of the best possible components. But, as with RAID, we have learned that we can obtain better reliability by combining multiple (very ordinary systems). If we have more computing and storage capacity than we actually need, we may be able to continue providing service, even after one or more of our servers have failed.

The key is to distribute service over multiple independent servers. The reason they must be independent is so that they have no *single point of failure* ... no single component whose failure would take out multiple systems. If the client and server instances are to be distributed across multiple independent computers, then we are building a distributed system.

Scalability

It is common to start any new project on a small system. If the system is successful, we will probably add more work to it over time. This means we will need more storage capacity, more network bandwidth, and more computing power. System manufacturers would be delighted if, each time we needed more capacity and power, we bought a new (larger, more expensive) computer (and threw away the old one). But

a. This is highly inefficient, as we are essentially throwing away old capacity in order to buy new capacity.
b. If we are successful, our needs for capacity and power will eventually exceed even the largest computer.

A more practical approach would be to design systems that can be expanded incrementally, by adding additional computers and storage as they were needed. And, again, if our growth plan is to *scale-out* (rather than *scale-up*) we are going to be building our system out of multiple independent computers, and so we are building a distributed system.

Flexibility

We may start building and testing all the parts of a new service on a notebook or desktop, but later we may decide that we need to run different parts on different computers, or a single part on multiple computers. If our the components of our service interact with one-another through network protocols, it will likely be very easy to change the deployment model (which services run on which computers). Distributed systems tend to be very flexible in this respect.

Challenges: Why are Distributed Systems Hard to Build

The short answer is that there are two reasons:

- Many solutions that work on single systems, do not work in distributed systems.
- Distributed systems have new problems that were never encountered in single systems.

New and More Modes of Failure

If something bad happens to a single system (e.g. the failure of a disk or power supply) the whole system goes down. Having all the software fail at the same time is bad for service availability, but we don't have to worry about how some components can continue operating after others have failed. Partial failures are common in distributed systems:

- one node can crash while others continue running
- occasional network messages may be delayed or lost
- a switch failure may interrupt communication between some nodes, but not others

Distributed systems introduce many new problems that we might never have been forced to address in single systems:

- In a single system it may be very easy to tell that one of the service processes has died (e.g. the process exited with a fatal signal or error return code). In a distributed system our only indication that a component has failed might be that we are no longer receiving messages from it. Perhaps it has failed, or perhaps it is only slow, or perhaps the network link has failed, or perhaps our own network interface has failed. Problems are much more difficult to diagnose in a distributed system, and if we incorrectly diagnose a problem we are likely to choose the wrong solution.
- If we expect a distributed system to continue operating despite the failures of individual components, all of the components need to be made more robust (eg. greater error checking, automatic fail-over, recovery and connection reestablishment). One particularly tricky part of recovery is how to handle situations where a failed component was holding resource locks. We must find some way of recognizing the problem and breaking the locks. And after we have broken the locks we need some way of (a) restoring the resource to a clean state and (b) preventing the previous owner from attempting to continue using the resource if he returns.

Complexity of Distributed State

Within a single computer system all system resource updates are correctly serialized and we can:

- place all operations on a single time-time (a total ordering)

- at any moment, say what the state of every resource in the system is.

Neither of these is true in a distributed system:

- Distinct nodes in a distributed system operate completely independently of one-another. Unless operations are performed by message exchanges, it is generally not possible to say whether a particular operation on node A happened before or after a different operation on node B. And even when operations are performed via message exchanges, two nodes may disagree on the relative ordering of two events (depending on the order in which each node received the messages).
- Because of the independence of parallel events, different nodes may at any given instant, consider a single resource to be in different states. Thus a resource does not actually have a single state. Rather its state is a vector of the state that the resource is considered to be in by each node in the system.

In single systems, when we needed before-or-after atomicity, we created a single mutex (perhaps in the operating system, or in memory shared by all contending threads). A similar effect can be achieved by sending messages to a central coordinator ... except that those messages are roughly a million times as expensive as operations on an in-memory mutex. This means that serialization approaches that worked very well in a single system can become prohibitively expensive in a distributed system.

Complexity of Management

In a single computer system has a single configuration. A thousand different systems may each be configured differently:

- they may have different databases of known users
- their services may be configured with different options
- they may have different lists of which servers perform which functions
- their switches may be configured with different routing and fire-wall rules

And even if we create a distributed management service to push management updates out to all nodes:

- some nodes may not be up when the updates are sent, and so not learn of them
- networking problems may create isolated islands of nodes that are operating with a different configuration

Much Higher Loads

One of the reasons we build distributed systems is to handle increasing loads. Higher loads often uncover weaknesses that had never caused problems under lighter loads. When a load increases by more than a power of ten, it is common to discover new bottlenecks. More nodes mean more messages, which may result in increased overhead, and longer delays. Increased overhead may result in poor scaling, or even in performance that drops as the system size grows. Longer (and more variable) delays often turn up race-conditions that had previously been highly unlikely.

Heterogeneity

In a single computer system, all of the applications:

- are running on the same instruction set architecture
- are running on the same version of the same operating system
- are using the same versions of the same libraries
- directly interact with one-another through the operating system

In a distributed system, each node may be:

- a different instruction set architecture
- running a different operating system
- running different versions of the software and protocols

and the components interact with one-another through a variety of different networks and file systems. The combinatorics and constant evolution of possible component versions and interconnects render exhaustive testing impossible. These challenges often give rise to interoperability problems and unfortunate interactions that would never happen in a single (homogeneous) system.

Emergent phenomena

The human mind renders complex systems understandable by contructing simpler abstract models. But simple models (almost by definition) cannot fully capture the behavior of a complex system. Complex systems often exhibit *emergent behaviors* that were not present in the constituent components, but arise from their interactions at scale (e.g. delay-induced oscillations in under-damped feed-back loops). If these phenomena do not happen in smaller systems, we can only learn about them through (hard) experience.