Travis Vaughn’s Study Log

04/02/2016

# What is a Distributed System

To me, the system will appear as one system working seamlessly to deliver results; however, behind-the-scenes it is actually a conglomeration of different computers working together.

This means distributed systems:

1. Consist of independently acting agents (ex. Computers)
2. Give appearance that client (human or machine) is only dealing with one system
   1. i.e. the multi-system interactions are hidden from the user
3. Should allow basically the same interaction each time regardless of where and when
   1. Think logging onto a computer in Chicago on a Monday and logging onto a different computer in Dallas on a Friday
4. Scaling should be pretty easy to complete
   1. Due to #3

Because of the 4 items above, distributed systems are often created as a middleware software between the applications and the operating systems. (That’s how it’s able to say “Hey, I see you’re on a linux box in NYC trying to access this web app ‘Learning to Love Distributed Systems.’ There’s this other person in San Fran accessing it from a Windows computer in order to learn with you and that’s no problem at all.”)

# When should we build Distributed Systems?

We don’t always need distributed systems. There are basically 4 important goals that need to be met in order to build a distributed system.

1. It allows easy access to resources
2. It is transparent
   1. Transparency here means seeming as one system to the client, not showing the user everything we are doing.
3. It is open
   1. It has standard rules to describe syntax/semantics of system
4. It is scalable

## Easy access to resources (to remote resources)

## Resources can be anything: computers, databases, Web sites, printers, etc.

Having easy access to remote resources is economical, provides collaboration (email, webex, shared docs, etc.),

Bad news: Security is severely lacking.

* I can buy as many “I Love Distributed Systems” tee-shirts as I want with John Doe’s credit card. There’s no proper identification required.
* Facebook can track all of my information, collect that data, and resell it to others so that now I’m getting a bunch of junk mail. Technically it’s legal in this case since I’ve agreed to their terms and conditions, but not every place is like this.

## Transparency

### What are some items the system should hide?

* Access
  + Data representation: Mac OSX file naming convention vs. Windows file name convention
* Location
  + Can’t tell where resource is physically located: URLs. <http://www.ilovedistributedsystems.com/index.html> does not display location of the main Web Server
* Migration
  + The URL also doesn’t say how long it’s been at its current location or when it moved.
  + This means we can move the physical location the main web server without affecting client access
* Relocation
  + Similar migration, but even better is that a user is currently using the system without any impact while resources are being moved
  + For instance, we have two wireless routers in my home, one for the front of the home and one for the back, when I move from one area to the other, I never notice that I’ve switched connections.
* Replication
  + Replication is the idea of placing a copy of a resource close to the place it’s accessed for performance purposes.
  + *Replication Transparency* deals with preventing the user from knowing several copies exist
  + It’s basically impossible to support replication transparency without also supporting location transparency
* Concurrency
  + Jane Doe accesses a website at the same time John Doe does. They both begin answering questions on the website that is then stored to a database. The system will appear to the users as if they are the only one using it, but technically the system is sometimes locking Jane’s request, until it’s finished processing John’s, but then it may lock John’s other requests while it finishes Jane’s. It happens so fast, that to both Jane and John, it appears like they’re the only ones in the system.
  + The locking allows the shared resource to be in a consistent state.
* Failure
  + A server failing doesn’t impact the client from completing their tasks (and the client has no idea a system has failed)
  + Hiding this is one of most difficult issues in distributed systems
    - Main difficult is telling the difference between a dead resource and extremely slow resource
    - Think about accessing a busy webserver and receiving a timeout request. The user doesn’t know if the server is actually down or not.

### When might transparency not be the best solution?

* If transparency affects performance too much
  + Hiding server failure before trying another one
    - May have been quicker to stop earlier and/or let the user cancel
* If replicas need to be consistent at all times
  + Think online gaming.
* May not be obvious that hiding distribution is a good idea
  + I don’t want to print a document at the Lincoln Park campus location when I’m in the Downtown location.

## Openness

Formalized protocols are generally specified through interfaces – typically refererred to as an **Interface Definition Language (IDL)**.

Typically only capture the syntax of the service. Thinking functions, parameters, return values, possible exceptions, etc.

### Proper Specifications

* Allows process that needs certain interface to talk to another process that provides that interface
* Two different people can build completely different implementations of those interfaces
  + Two different distributed systems that operate the same way
* Are complete and neutral
  + Completeness means everything necessary to implement
  + Neutral means doesn’t describe what implementation should look like
  + These are important for interoperability and portability
    - Interoperability: how two different implementations of systems work together by merely relying on other’s services as specified by the standard.
      * The example I can think of at work is how to access my work’s corporate actions database. There is a service built on top of the database that multiple other services can access (we have trader’s programs accessing, autoposters accessing, etc.). These systems that interact with the corporate actions service also pass along information from the service between each other.
    - Portability: how to move an application developer for system A to system B using same interfaces as A.
* AN IDL CAN BE CREATED WITH A PEN AND PAPER

System should have ability to be configured out of different components and be easy to add new components or replace other components without affecting the components already in place. We call this being **extensible**. For example, adding parts that allows parts to run on different OS.

Separate Policy from Mechanism: add more notes here when I understand it better.

## Scalability

Three ways scalability can be measured

1. Size
2. Geography
   1. What about when classes become global
3. Admin (often not discussed, but causes some of the most problems)

Bad news: As we scale up in one of these departments, we typically lose performance

### Scalability problems

* Size
  + When need more users/resources, confronted with limitations of centralized services (single server for all users), data (single database for all employee records), and algorithms (routing based on complete information).
    - Think services centralized on one server on one machine
    - Sometimes need centralized locations: confidential information
  + Only decentralized algos should be used:
    - No machine has all info on system state
    - Machines make decisions based only on local info.
    - Failure of one machine doesn’t impact others
    - No global clock
      * Saying at exactly 07:00:00 all machines will shutdown will fail because it is impossible to get all clocks exactly synced.
  + Centralization is should be avoided whenever possible
  + Examples: How many students can fit in an online classroom?
    - Bandwidth
    - Room in grading links
* Geography
  + When scaling distributed systems designed for LANs, tough because based on synchronous communication. On WANs, those milliseconds for interprocessing add up over thousands-millions of users.
  + Communication in WANs is unreliable and basically always point-to-point whereas LANs are typically reliable based on broadcasting
    - Example: on a LAN, it can send a message to all connecting services if it’s using service. Only machines that have service respond. This would be awful on a WAN, instead they would need to use special location services
  + Strongly related to problems of centralized solutions
    - Centralized components waste network resources
    - Thinking about playing a game with your buddy online, but every action you take has to first be routed to a server in Hong Kong and come back. The game would be severely lagged.
  + Examples
    - Timing problems: deadlines in the middle of the night for some students
    - Are certain content or components illegal in other areas?
      * Encryption is considered “munitions” (like a gun) in some countries
* Admin
  + Conflicting policies is always a major problem
    - Resource usage and how to pay for it
    - Management
    - Security
  + Examples:
    - How to give 4,000 students grades?
    - Where are assignments stored?
    - Hierarchy of instructors
      * How do they coordinate?

### Scaling Techniques

Most problems show up as performance issues caused by limited capacity of servers and networks.

Three basic techniques for scaling

1. Hiding communication latencies
2. Distribution
3. Replication

#### Hiding Communication Latencies

Avoid waiting for responses to remote (and distant) service requests where possible. Example, when client is waiting for response from server, allow requester to do other useful work instead of waiting. When reply comes in, requester is interrupted and a special handler is called to complete previously issued request. Basically constructing **asynchronous communication,** which can often be used in batch-processing systems parallel applications. (Think of automated Tidal jobs at work that run at midnight)

Where asynchronous communication is not possible, try shipping some service tasks to client (such as providing database user-forms and checking for accuracy). This is what is typically down now on the Web in the form of Java applets and Javascript.

#### Distribution

Takes a component, splits it into smaller parts, and spreads those parts across the system.

Example: Internet Domain Name System (DNS)

* Organized into a tree of domains
* Domains are divided into mutually exclusive zones
  + int, com, edu, gov, mil, org, net, jp, us, etc.
  + Sub categories under each for names: yale(.edu), etc.
* The names of each one are handled by a single server

Another example is World Wide Web

#### Replication

Increase availability and balances load between components (which causes better performance!). In global systems, having a copy nearby can hide communication latency problems

Caching is a special form of replication. Like replication, caching makes a copy of resources, but unlike replication, the decision is isolated to the client of the resource, not the owner of the resource. Also, caching is on-demand as opposed to where replication is planned ahead of time.

Bad news: Updating a copy leads to potential inconsistency in other copies. Not an issue for say a news article from a website that a user is reading, but hasn’t been updated for a couple of days, but imagine not getting an update on a stock ticker for a couple of minutes (or a couple of milliseconds in the case of high-frequency trading!).

The problem with strong consistency is updating all of the copies immediately. Sometimes in a specific order (if updated concurrently)

#### Considering Scaling Techniques

Size scalability may be less technically problematic technique. Mother Nature is typically at play with geographical scaling. Admin scaling is typically **MOST DIFFICULT** (nontechnical issues are a big part). Peer-to-Peer technology can only be a partial solution (best-case scenario).

## Pitfalls

False assumptions typically made when creating distributed application:

1. Reliable network
2. Secure network
3. Homogenous network
4. Topology doesn’t change
5. Zero latency
6. Infinite bandwidth
7. Zero transport cost
8. One administrator

# Types of Distributed Systems

1. Distributed Computing Systems
2. Distributed Information Systems
3. Distributed Pervasive Systems

## Distributed Computing Systems

Two sub groups:

1. Cluster computing
   1. A bunch of PCs, closely connected by high speed LAN
   2. Each node runs same OS
2. Grid computing
   1. Each system may fall under different admin domain
   2. May be very different when it comes to hardware, software, and deployed network tech.

### Cluster Computing Systems

Became popular when price/performance ratio of PCs improved. Essentially build a super computer using basic ready-made tech by simply connecting these simple computers in a high-speed network.

Cluster computing is basically only used for parallel computing where one program runs at the same time on multiple machines.

Examples of cluster computing

* Beowulf clusters (hierarchical)
  + Master-Compute model
  + Master node controls and accesses a collection of compute nodes in the cluster collection
  + Master assigns nodes to a particular parallel program, maintains batch queue of submitted jobs, and provides interface for the users of the system
  + Master runs middleware for execution of programs and management of cluster
  + Compute nodes need nothing else but a standard OS
* MOSIX (systematic)
  + Attempts to provide single-system image of cluster
    - Means to a process, the cluster computer appears as a single computer (distribution transparency)
  + Dynamically and preemptively migrate processes between nodes that make of cluster
    - Process migration allows user to start an app on any node
    - App can transparently move to other nodes (ex. to free up resources)

### Grid Computing Systems

Heterogeneous: no assumption of hardware, OS, networks, admin domains, security policies

Typically resources consist of compute servers (potentially even cluster supercomputers), storage facilities, databases, and special networked devices (think telescopes, sensors, etc.).

Most of software revolves around different admin domains. For this reason, focus is often on architectural issues.

Possible architecture:

Applications 🡪 Collective layer

🡪 Connectivity layer

🡪 Fabric layer

🡪 Resource layer

🡪 Fabric layer

Fabric Layer:

* Provides interfaces to local resources at a specific site
* Typically will provide functions for querying state and capabilities of resource…think locking resources.

Connectivity layer:

* Communication protocols for supporting grid txns that span use of multiple resources
* Need protocols because how else will systems know how to communicate data
* In many human cases, users aren’t authenticated instead programs acting on behalf of users are authenticated. This delegation of rights is important function that must be supported in this layer.

Resource layer:

* Responsible for managing a single resource and responsible for access control
* Uses functions provided by connectivity layer and calls directly the interfaces made available by fabric layer
* Relies heavily on authentication performed in connectivity layer
* Example: This layer offers config. info functions on a specific resource, or to perform specific ops such as creating a process or reading data

Collection layer

* Handles multiple resources and typically consists of services for resource, discovery, allocation, and scheduling of tasks onto multiple resources, data replication, and so on
* May consist of a bunch of protocols for many different purposes (dissimilar from resource and connectivity layer which are relatively small)

Application layer

* Consists of applications that operate within virtual organization and use grid computing environment

Middleware of grid typically consists of collective, connective, and resource layer. The notion of a site (admin unit) is common.

## Distributed Information Systems

1. Transaction Processing System
2. Enterprise Application Integration

### Transaction Processing System

Practically, ops on a DB are usually carried out in form of a Tx. This requires primitives supplied by underlying distributed system or language runtime system. These primitives depend on system (example: mail system might have send, receive, and forward mail.)

Remote Procedure Calls (RPCs), calls to server, are typically encapsulated in Tx (called transactional RPC).

Properties of Transactions

1. **A**tomic: appear to happen indivisibly
   1. All txns happen or none happen
2. **C**onsistent: does not violate system invariants
   1. Example: Poker game states all money must stay on table. After each action, the amount of money transferred from one player to another or to the collective “pot” must not cause a difference to the overall amount of money on the table.
3. **I**solated: concurrent transactions don’t interfere with each other
   1. To each other and others final result looks as though al txs are sequentially in some order
4. **D**urable: when transaction commits, changes are permanent
   1. No failure after the commit can change the transaction

Typically referred to as ACID

Nested transaction:

* Consists of subtransactions (example going to an airline database vs. a hotel database)
* Give rise to subtle, but important problem: parent commits one of subtransactions before others finishes, making it visible to parent, but then parent has to abort. The results of the committed transaction must be undone and thus permanence only applies to top-level transactions
* If subtransaction commits and then second one starts, second one will see results produced by first

### Enterprise Application Integration