Study Log of Distributed System

1. Definition
   1. What is distributed system?

A distributed system contains two of more independent computers/machines/servers. They may share some data, keep communication with each other, and act as uniformly to the client. To the outside world, they are treated as a single system. The client even doesn’t know there may be hundreds and thousands of machines behind the user interface.

* 1. Features for distributed system
     1. Complexity

Compared with single node system, a distributed system is more complicated. For example, the data may not be able to be fetched from local, you have to communicate with other machine. And other challenges like how to synchronize time, how to share data, how to replicate data. Finally, how to make it act as a single machine to the client, along with keep being correct, stable and robust.

* + 1. Multi-task support

A distributed system shall have the ability to work on multiple tasks simultaneously. This is the main advantage why we use it instead of a standalone system.

* + 1. Robustness

Another advantage of distributed system is robustness, compared with the single node system. If a machine of a distributed system is shutdown or dead, it will be abandoned. System will find a substitute machine to continue handle the task.

* + 1. Uniform interface

Though, the structure of a distributed system may be very complicated. However, the inside components must be packaged as a whole unit. All of the internal components are put into a black box. When we are using it, it is not necessary to know how its components cooperate with each other. We just need a uniform interface and the identical inputs and outputs.

* + 1. Transparency

Whatever the architecture of the distributed system is and however the complexity of the system, it is always transparent to the end user. Transparency includes following aspects:

* Access – How a resource is accessed is unknown to end user. The user just care about he/she the result that the resource is fetched.
* Location – Where the resource is located is unknown to end user. There may be lots of duplicated data in several machine or location. To the end user, he/she just gain one of the copy, and it doesn’t matter, where it comes from.
* Migration – The resources may be moved to other places, but the end user doesn’t realize and he/she actually doesn’t need to know that, since the system still works as nothing has been changed.
* Relocation – Sometimes, the resource the user is asking for may not be available from the original location. So the system may relocate the user to another machine or sub system to get it. The end user has no idea about the details about this procedure.
* Replication – Generally, a distributed system create several duplicated data in different machines. This is so-called replication. The data the user is operating on may be replicated and the end user doesn’t know.
* Concurrency – Since a distributed system can handle multiple tasks simultaneously, it is often happened that one resource may be accessed by several users at the same time. The distributed system must solve the conflict problem, keep the atom transaction for each user and prevent from creating dead lock. For each end user, he/she doesn’t realize the detailed situation, it seems only one person is operating on the resource.
* Failure – Error/exceptions are hard to completely avoid. When a failure occurs, the task would be executed again on another machine or at another time. To end user, he/she totally has no idea what has happened, since the success result will still be returned to him/her.
  1. Interface Definition Language

IDL is used to define uniform interfaces for the distributed system to let others access. Generally, it at least has two arguments: one is the length of the data; another is actual data contains a string with that length. Relevant terms: Open system, IDL, Client Stub, Server Skeleton(Stub).

* 1. Distributed Algorithm

When designing a distributed system, we actually are design a distributed algorithm. And we must know that there are some limitations for a distributed system.

* Each node/component has only local information but no whole picture about the state
* Each node makes decision based on its limited partial knowledge
* One machine’s failure doesn’t affect to others or ruin the whole system.
* No global clock exists.
  1. False Assumption
* Network is reliable
* Network is secure
* Network is homogeneous
* Topology doesn’t change
* Latency is zero
* Bandwidth is infinite
* Transport cost is zero
* There is an administrator
  1. Transaction
     1. Characteristics
* Atomic: The transaction’s inner is black box, no matter how it is complicated.
* Consistent: After execution, the balance of constants is not broken. Resources/values are just moved from only place to another, no disappear, no new objects from nothing.
* Isolated: No disturbance between concurrent transactions.
* Durable: Changes becomes immutable once the transaction is confirmed.
  + 1. Test and Set
* User ‘Test and set’ to preserve atomic operation in local machine.
* Hard to use ‘Test and set’ for remote machine, which has different memory spaces.
* Hard to implement shared memory to make distributed system coordinate well.

1. Architecture
   1. Architecture Styles

* Layered: One layer can only communicate with the nearest layer. Suppose a new request comes, it is handled by the first layer. Then, the first layer calls to the next layer. The layers are invoked one by one from top to down. Later, the deepest layer finishes the work, and it calls the upper layer. The result is transferred back to the top layer.
* Object-based: Each component in the distributed system is abstracted to object. Each object has its own properties and functionalities. One object may have no, one or multiple relationships with others. These relations are created depends on the business model and your design. With these relationships, one object can call another object to get a specific result, invoke an event, or just notify others.
* Data-centered: In this system, all of the data is stored in a centralized location. Each component needs to contact the data center to get the required information. The benefits of this style is, no synchronization is required, since there is only one data, no copy. If we are building a heavy server, we can consider to use this style. The disadvantage is obvious, the system will collapse if data center goes down. So it is recommended to have another shadow data center which can take the responsibility immediately if the data server goes down.
* Event-based: This is also known as Publish-Subscribe mode. The components are not tied together. Each component can be a publisher, meanwhile, it can be a subscriber as well. The publisher will invoke the event and notify other components which have subscribed the event. The subscriber received the notification and take the corresponding actions. The benefits of this style is, components are decoupled with each other. It is easy to create, update or remove component without or only a little effects to the whole system.
  1. Application Layer
* User-interface Level: The responsibility of this level is to accept input and display output. The layer the interface between end-user and system. It can be a webpage, a console or a touchable screen.
* Processing Level: This is core of the system. All of the logical functionalities are located in this level. It is the brain of the system. It receives the request from the user-interface process it and send the result back. It may contains several components, each of them has a specific responsibility. In the reality, it can be a web server or a cloud service.
* Data Level: For data level, it is not so mysterious. It just stores data and provide the data service to the processing level. It can be a separated and independent system. Though itself can be complicated, however, to the process level, it is a black box, just returns the data that the processing is asking for.
  1. Alternative Client/Server Organization
     1. Heavy Client

Client contains all of the user-interface and application layers, even parts of database. It does most of the job. To the end user, it reacts quickly and the experience using it is great, no latency. To the server, there is less communication through network. However, it has some disadvantages. More work required if a new version is released. All of the clients need to be updated. Some tricky problems may occurs because of the different versions. And the system administrator has to pay more attention to maintain the system and keep it running properly. If any error/exception occurs, client has to handle it locally. And it is also hard for programmers to diagnose cause of issue. Because each client may be running in a special environment. It may be hard to reproduce the issue and impossible to debug at client side.

* + 1. Heavy Server

All of the database and application layers are in the server side. Some parts of the user-interface layer can also be put into the server. The biggest benefits is that no need to distribute new releases to all of the clients, just need to deploy to the server for once. The source codes are easy to maintain. The maintenance fees are also lower. Support packages for the new features and bug fixing can be delivered quickly and smoothly. Only the server need to be upgraded. There are also some disadvantages. First, there are much more communications between clients and server. Latency may occurs, and the user experience is not so good. Second, since most of the processing layer is in the server, it becomes a bottleneck. It is required that more cost is spent on the server. And the server becomes more complicated to be stable, robust and efficient. Third, the server may store some sensitive data for the client. So, security is another issue.

* + 1. Choice

Each approach discussed above has its own advantage and disadvantage. It’s hard to say which one is better. And there is no uniform standard to choose heavy client or heavy server. All depends on requirements or scenarios. Or merge and combine them in a large scale distributed system. For example, if we are building a Cloud service which provide weather forecast, we may choose heavy server. Since we don’t know how many users will connect to our server. We just setup a server with all functions there and publish a public API for user to call. Another example, if we are building an online game, we may choose heavy client. For a game, it shall be installed in end-user’s machine. This client can provide fantastic animations and sound. The client only connects to the server when necessary.

1. Processes
   1. Process and Thread

These two concepts come from operating system. Process is a smallest unit, which can obtain resources(cpu processing time, memory) and handle requests. For any application, there must be at least one process. Threads are embedded into process. They are workers for process. They cannot own resources by themselves, instead, they share the resources with other threads, which are all belong to their parent process. Processes can use IPC(InterProcess Communication) to communicate with each other.

* 1. Virtual machine

Virtual machine is an interface above some layer or system. It contains specific machine instructions, invoke them and simulates as another system. It can locates above the operating system or under it.

1. Communication
   1. Protocol Level
      1. Protocol Layer

* Application: High level, focus on functionality.
* Presentation: Different data form, texts, pictures, videos.
* Session: Setup conversations between application, maintain their states.
* Transport: Wrap the transportation process, make sure data is transferred successfully.
* Network: It’s about routing. How to find an efficient path to transfer packets from the departure node to the destination node.
* Data Link: Basic data format, how to encode or decode packet.
* Physical: Facilities or hardware used to transfer data, eg, cable, light, signal.
  + 1. UDP
* One way, from sender to receiver packet
* Low cost, efficient, without setup
* But no guarantee of the transportation, the sequence of receiving may be different from sending, data may lost
  + 1. TCP
* Two way, each node can send or receive packet
* Reliable, packets in order
* Costly, setup is required
* Buffer is required, may delay
  1. Sync/Async Calls
     1. Synchronous Call
* Executed in sequence, easy to implement
* Call will wait for the response, block may happen
* Inefficient because of wait
  + 1. Asynchronous Call
* Executed not sequentially, much more complex
* Implemented with multi-thread
* Main process won’t hang, no wait
* More efficient
* Hard to debug
  1. Socket/Port
     1. Socket

Socket is a low layer implementation of server or client. The concept of server or client is relative. A server can be client to other servers. A server socket is the one which provides services via receiving request and sending packets back. A client socket is the one which ask for services via sending request and receiving feedback. To socket itself, it doesn’t care. The job for it is just to send or receive data packets. Each socket has an IP address and a port.

* + 1. Port

Port is a channel for transferring packets. Each socket has a specific port number. There can be multiple sockets in one server. They may have different ports, so they can work on different channels. We can define one port for TCP communication, another port for UDP communication. Both of them won’t disturb each other. So this makes it possible that one server can be setup with the capability to handle multiple tasks.

* 1. Data Transfer
     1. Data Format

Different systems may have different data formats. When they try to communicate with each other, a unified data format is necessary for transportation. So the receiver can understand and consume the data.

* + 1. Marshalling/Unmarshalling

Marshaling is process that the sender convert any object or private data format to other unified format, like xml, before sending data out. XML is common standard and is widely used. Unmarshaling is the opposite procedure to convert XML data to objects or other format which the receiver can consume.

* 1. Remote Procedure Call
     1. Steps of RPC

Client->Local OS->Remote OS->Server->Local OS->Client OS->Client

* + 1. RPC Exchange Protocols
* R
* RR
* RRA

1. Security

Encryption

Certificate

1. Assignment
   1. JokeServer
      1. JokeServer

* JokeServer holds the data for joke/proverb list, and also holds the identities and their states of the clients.
* Joke Server provide two services. One is to handle joke/proverb request, another is for admin purpose. These two services are implemented by socket with different ports.
* When a joke/proverb request comes, JokeServer dispatch the task a Worker in separate thread.
* When JokeServer is starting up, an AdminListener service is started up as well.
  + 1. Worker
* A worker is one-time working unit. It is created by JokeServer whenever a new joke/proverb request comes. After finishing the task, it will disappear.
* There may be multiple Workers running at the same time, they work simultaneously without interfere with other workers during processing.
* Worker gets the request and other relevant information from JokeServer, generates a joke or proverb(depends on the current server mode), sends back to client.
  + 1. AdminListener
* AdminListener works on a separate thread and only monitors at the specific admin channel(port).
* When a new admin request comes, AdminListener dispatch it to AdminWorker, similar with how JokeServer works. It will create a new instance of AdminWorker.
  + 1. AdminWorker
* Like Worker, the AdminWorker is also a one-time working unit. After finish handling the admin request, it will disappear.
  + 1. JokeClient
    2. JokeClientAdmin
  1. MyWebServer
  2. Mimer
  3. HostServer
  4. AsyncJoke
  5. Distributed Intelligent Agent

1. sdfsd