A Framework Of The Distributed Server In Actor Model

Author: peimin
Email: peimin@outlook.com

content

1.Introduction

Nowadays concurrent programming has become a hot topic in computer science. Languages like Erlang, Go and Scala are gaining extensive attention due to their excellent support for concurrency. The driver force behind this phenomenon is known as the "multicore crisis[x]."

Moore’s law[x] continues to increase the number of transistors per chip, but we’re seeing CPU with more and more cores rather than a single CPU become faster by utilizing those transistors. The Symmetric multiprocessing became the typical architecture of CPU.

As Herb Sutter said, “The free lunch is over.”[x] It stated that the speed of microprocessor serial-processing is reaching a physical limit, which means that the products with multi-core cores will be focused on by processor manufacturers. The most important things for software developers is that they will must develop tons of concurrent and paralleled programs to exploit multiple cores. This paper will focus primarily on the development of concurrent programs.

Concurrency is about a great deal for computer systems, it allows your software to be responsive, fault tolerant, efficient, and simple. For development of software, there are seven typical models of concurrent programming[x].

Threads and locks:
Although threads-and-locks programming has many problems, it is still the default choice for many concurrent applications.

Functional programming:
Functional programming with excellent support for concurrency is becoming increasingly prominent as they eliminate mutable state that means intrinsically thread-safe .

The Clojure Way—separating identity and state:
For concurrency, the approach to state of Clojure is characterized by the concept of identities that are represented as a series of immutable states over time.

Actors:
The actor model with widespread applicability is a general-purpose concurrent

programming model. It provides particularly strong support for fault tolerance and resilience.

Communicating Sequential Processes:
Communicating Sequential Processes (CSP) has much in common with the actor model, both being based on message passing. Its emphasis on the channels used for communication and CSP processes are anonymous, while actors have identities.

Data parallelism:
You have a supercomputer hidden inside your laptop. The GPU utilizes data parallelism to speed up graphics processing, but it can be brought to bear on a much wider range of tasks. There is some framework for GPU programing and computation such as CUDA that is a parallel computing platform and programming model invented by NVIDIA and OpenCL™ (Open Computing Language) that is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms..

The Lambda Architecture[x]:
The Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both the strengths of MapReduce[x] and stream-processing methods.

When it comes to those concurrent models, threads-and-locks programming is extensively used in highly concurrent applications. But unfortunately, there have been a high number of accidents that have happened in past years because of the problems with multithreading programming applications.

For example, the accident of Mars Pathfinder in terms of transfering data occured in 1997s[x], th accidents of Therac-25[x] between 1985 and 1987 because of concurrent programming errors and so on. The reasons of those accidents is that there are some criticism of multithreading concurrent paradigm[x]:

- performance: Many software used threads and lockc have not performed well.
- Control Flow: Threads have restrictive control flow.
- Synchronization: Thread synchronization mechanisms are too heavyweight.
- State Management: The managements of live state with thread stacks are an ineffective way.
- Scheduling: The scheduling of threads is nondeterministic.

The above arguments show that why thread-based programming is primitive, difficult to drive, and both unreliable and dangerous compared to newer technology. Compared with thread-based programming, Actor Model[x] is a more completed model forconcurrent applications. Actor Model uses message-passing to communicate with each others instead of shared memory with locks that maybe cause deadlock and potentially another problems. One the one hand, it targets both shared-memory and distributed-memory architectures while On the other hand, it facilitates geographical distribution and provides especially strong support for fault tolerance and resilience. Hence, the actor model among those concurrent models is more reasonable for game services or web services.

In this paper, we will focus on actor model and design a distributed concurrent server in actor model for game services, web services and so on.

2.Actor Model

2.1 History Of Actor Model

The Actor model is different from other concurrent models, it was influenced by physics such as general relativity and quantum mechanics, the programming languages Lisp, Simula and early versions of Smalltalk, capability-based systems, packet switching and so on[x].

In 1975s, Irene Greif developed an operational semantics for the Actor model in his dissertation "Semantics of Communicating Parallel Processes[x]" in MIT. In 1977s, Henry Baker and Hewitt published a set of axiomatic laws for Actor systems in their dissertation "Laws for Communicating Parallel Processes[x]" in IFIP.

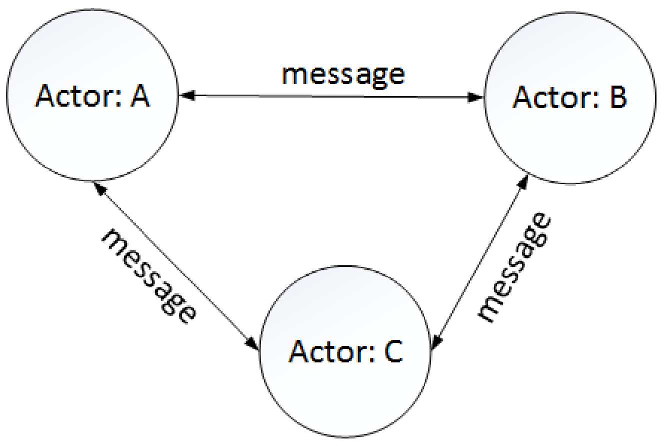
Other major milestones include William Clinger's 1981 dissertation introducing a denotational semantics based on power domains and Gul Agha's 1985 dissertation named " A Model of Concurrent Computation in Distributed Systems", which further developed a transition-based semantic model complementary to Clinger's. This resulted in the full development of actor model theory.

Major software was implemented by the Message Passing Semantics Group at MIT. Further the message passing in the model was developed by Research groups led by Chuck Seitz at Caltech and Bill Dally at MIT.

2.2 What is Actor model?

Actors are objects that encapsulate state and behavior[x], they communicates with each other by messages-passing. In a sense, An actor is similar to an object in an object oriented program like java or C++. The difference is that actors are concurrent instead of just calling a method to exchange messages in OO-style, actors communicate by sending messages to each other`s mailbox. The system in actor model can been in more detail in Figure 1.

Figure 1. system in actor model.



The philosophy of Actor model is that everything is an actor and an actor is a service or a computational entity, which is similar to the everything is an object philosophy in OO programming languages. While the Actor model is inherently concurrent rather than executed sequentially in object-oriented software.

The difference between the actor model and object-oriented programming is that the messages sent by actors is asynchronous without blocking for the messages finished, while the messages sent by objects in object-oriented programming is synchronous and the communication among the actors is just by used message-passing to cooperate with each others instead of shared memory used by functions in object-oriented programming to communicate.

In distributed system, every process is deem as a actor, they communicate with each others by message-passing instead of shared memory. In most system, the actor is a abstract object that communicate with each others by message-passing and work independently without influence by others.

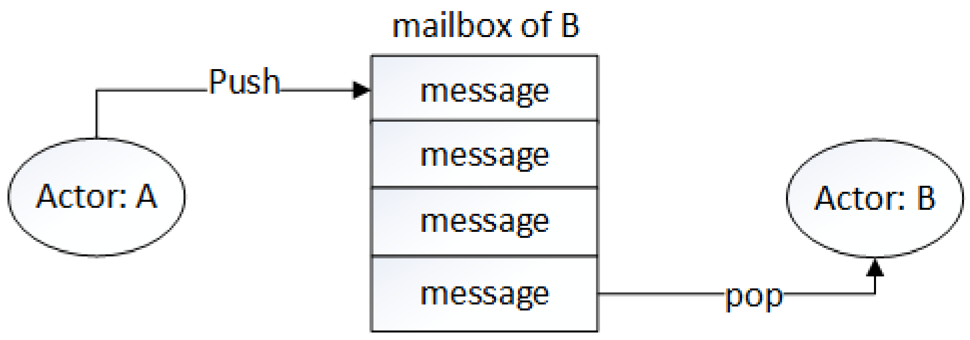
The advancements of the Actor model include inherent concurrency, dynamic creation of actors, interaction only through direct asynchronous message-passing without restriction on message arrival order, decoupling the sender from communications, light weight objects and so on.

2.3 Mailbox

The purpose of an actor is to process messages from other actors or from remote the actor system[x]. The piece between senders and receivers is the actor's mailbox: each actor has exactly one mail address and one mailbox all senders enqueue their messages to. Enqueuing happens in the time-order of send operations, which means that messages sent from different actors may have a undefined order at runtime due to the apparent randomness of distributing actors across threads. On the other hand, Sending multiple messages to the same target from the same actor will enqueue them in the same order.

There are tons of implementations of mailbox, FIFO(First In, First Out queue) is usually adopted, which means that the order of the messages processed by the actor matches the order of they were enqueued. Sometimes, priority queue may be used to prioritize some messages over others. In this way, a priority mailbox will enqueue not always at the end but at a position as given by the message priority. The mailbox in the actors can been in more detail in Figure 2.

Figure 2. mailbox in the actor:



2.4 Asynchronous and non-blocking

In Actor model, sending and processing a message is tackled in an asynchronous and non-blocking way. When the sender send a message, they can immediately continue with own work instead of blocking until the message has been processed by the receiver. Because what really happens when sending a message to an actor is that placing this message in the recipient`s mailbox, which is a non-blocking operation.

The actor would blocks the thread as long as it takes to process the message, which means that lengthy operations degrade overall performance, as all the other actors have to be scheduled for processing messages on one of the remaining threads. In this case, the principle for recepient`s functions is to spend as less time as possible. Most importantly, avoid calling blocking code inside your message processing code, if possible at all.

Furthermore, there is something you can’t prevent doing completely that the major drivers of database such as redis is still blocking, for querying and persisting data for it from the actor-based application. There are solutions to this dilemma, but we won’t cover them in this paper.

2.5 Actor Best Practices

•Actors should cooperate with each other without bothering others. Implementation in programming means to process events and responses (or more requests) in an event-driven manner. Actors should not block such as occupying a thread all the time. On some external situations, a lock may be not prevent like a network socket, another thread should be used to do this.

• Do not exchange mutable data in message-passing. If the encapsulation of actors is broken by exposing their mutable state to the outside, you are back in normal threads-and-lock concurrency.

• A single thread implement the management of Blocking. In some cases it is unavoidable to do blocking operations. Dedicate a single thread to manage a set of blocking resources (e.g. a epoll on oone thread in linux driving multiple channels) and dispatch events as they occur as actor messages. For database, common pattern is to create a router for N actors, each of which wraps a single DB connection and handles queries as sent to the router.

2.6 Programming with Actors

A lot of different programming languages support the Actor model. These languages include: Lua Erlang Scala C++.. Actor libraries or frameworks have also been implemented to permit actor-style programming in languages that don't have actors built-in. Among these frameworks are:

Name	License	Languages
Actor Framework	Apache 2.0	.Net
Akka	Apache 2.0	Java and Scala
Cloud Haskell	BSD	Haskell
ActorKit	BSD	Objective-C
Theron	MIT	C++
Pykka	Apache 2.0	Python
Libactor	GPL 2.0	C
Celluloid	MIT	Ruby
Orleans	MIT	C#

2.7 summary

The Actor model is to avoid all the problems of threads-and-lock model and allowing you to write highly performant concurrent software. It requires you to design and write your application from the ground up with concurrency.The idea is that your application consists of lots of light-weight entities called actors. Each of these actors is responsible for only a very small task and cooperate with each other by message-passing. In program languages, it always uses coroutines to implement actor model, in last chapters we will inroduce coroutines and desgin a dsitributed server in actor model.

3 coroutines

A coroutine is similar to a thread. It has private stack, private local variables, and private instruction pointer; but sharing global variables and mostly anything else with other coroutines.Coroutines allow multiple entities to suspend and resume execution at certain locations to generalize subroutines for nonpreemptive multitasking. Coroutines are convenient for implementing more familiar program components such as cooperative tasks, event loop, iterators and pipes.

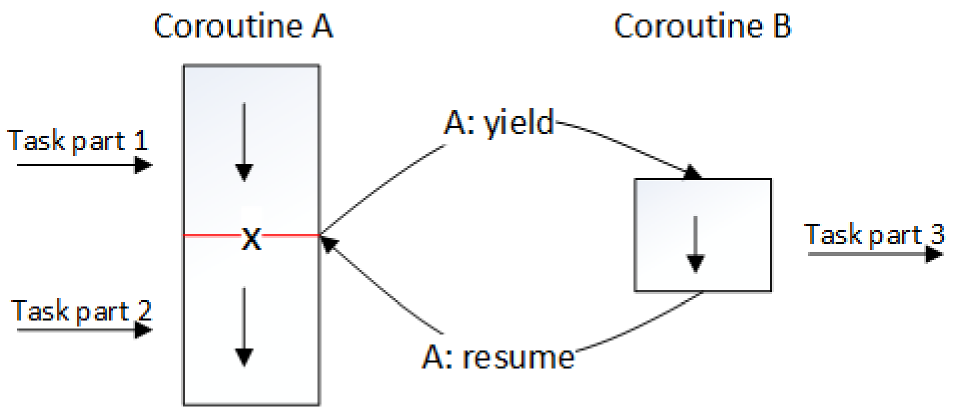
3.1 coroutines vs threads

The difference between threads and coroutines is that threads are preemptively scheduled while coroutines are not. Programs with threads-and-locks model must be careful as threads can be rescheduled at any instant and can execute concurrently. By contrast, programs using coroutines can often avoid locking entirely due to the schedule of coroutinesis nonpreemptive. This property is deemed as a benefit of event-driven or asynchronous programming.

3.2 comparison with subroutines

A coroutine instance holds state, and varies between invocations, while subroutines donot hold. Calling another coroutine by means of "yielding" to it and then would return to the point where they were invoked in the original coroutine. From the coroutine's point of view, it is not exiting but calling another coroutine. The cooperation between the coroutines can been in more detail in Figure 3.2.

Figure 3.2. a example of coroutine yield and resume.



In this example, the task is comprehensive of three parts. when task part 1 is finished up to position of x, yield function of coroutine A is invoked, and then A suspend and coroutine B run. After B is finished, resume function coroutine A is invoked and A continue to run from postion of x until it is finished. It means that coroutine A cooperate with coroutine B to finish the task.

3.3 common uses

There are some useful implementations with coroutines:

- State machines
- Actor model of concurrency.By using coroutines in programs, actors can executes procedures sequentially and continuously.
- Communicating sequential processes(CSP) model of concurrency where each sub-process is a coroutine.

3.4 Implementations

Coroutines are supported Early in Simula and Modula-2. In recent years many of the most popular programming languages including Ruby, Lua, Erlang, C++, and Go have supported it.

3.5 coroutine in lua

Lua offers all its coroutine functions packed in the coroutine table. The create function creates new coroutines. A coroutine can be in one of three different states: suspended, running, and dead. When we create a coroutine, it starts in the suspended state, which means that a coroutine does not run its body automatically when we create it. We can check the state of a coroutine with the status function.

The function coroutine.resume (re)starts the execution of a coroutine, changing its state from suspended to running.The real power of coroutines stems from the yield function, which allows a running coroutine to suspend its execution so that it can be resumed later. Let us see a simple example:

```
1    local co = coroutine.create(function ()
2         for i = 1, 3 do
3                  print("co", i)
4                  coroutine.yield()          -- suspend
5         end
6    end)
7
8    print(coroutine.status(co))          -- print: suspended
9    coroutine.resume(co)                 -- print: co 1
10   print(coroutine.status(co))          -- print:suspended
11   coroutine.resume(co)                 -- print: co 2
12   print(coroutine.status(co))          -- suspended
13   coroutine.resume(co)                 -- print: co 3
14   coroutine.resume(co)                 -- nothing to print
15   print(coroutine.status(co))          -- print: dead
```

From line 1 to line 6, we create three coroutines and suspend them. In line 8, we check its status, we can see that the coroutine is suspended and therefore can be resumed again.In line 9, when we resume this coroutine, it starts its execution and runs until the first yield. When we resume the coroutine, this call to yield finally returns and the coroutine continues its execution until the next yield or until its end.During the last call to resume in line 15, the coroutine body ended the loop and then returned, so the coroutine is dead now.

consumer-producer using coroutine in lua

There is a simple example of consumer-producer problem to explain the importance of coroutines:

producer

```
function producer()
       return coroutine.create(function()
              while true do
                     local val = io.read()    -- producer get new value
                     coroutine.yield(val)
              end
       end)
end

function filter(prod)
```

```
        return coroutine.create(function()
                for line = 1, math.huge do
                     local status, val = coroutine.resume(prod)
                        val = string.format("%5d %s", line, val)
                        coroutine.yield(val)      -- send value to consumer
                end
        end)
end

function consumer(prod)
        while true do
              local status, val = coroutine.resume(prod)
                io.write(val, "\n")
        end
end

prod = producer()
filt = filter(prod)
consumer(filt)
```

In this example, when the producer coroutine recevie data from IO, it is suspended by call yield function and sends data to the consumer coroutine. When a consumer coroutine receives data from producer coroutine, it will print it and then suspend by call resume function to recevie data from IO. Loop it.

3.6 summary

The coroutines with private stack, private local variables, and private instruction pointer allow multiple entities to suspend and resume execution at certain locations to generalize subroutines for nonpreemptive multitasking. By using coroutines in programs, actors can executes procedures sequentially and continuously.

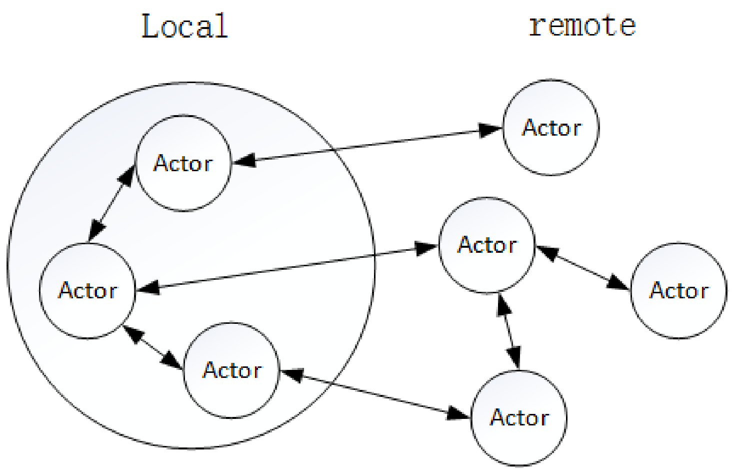
4 The design of a distributed server

The computer systems for game services and web services are usually comprised of more than one computer as those systems are inherently concurrent. In this case, the distributed server system is crucial for those systems. In this chapter we will desgin a distributed server system which not only utilizes efficiently multiple cores and cluster of cumputers, but also makes software development more easier.

4.1 abstract of systems

In this system, every thing is deemed as an actor wherever it is located in local computers or remote computers and an actor is a service that maybe a game service in game server system or a web service in web system. It depends on your system. Actors communicate and cooperate with each others by message-passing.The abstract of system in the actor model can been in more detail in Figure 4.1.

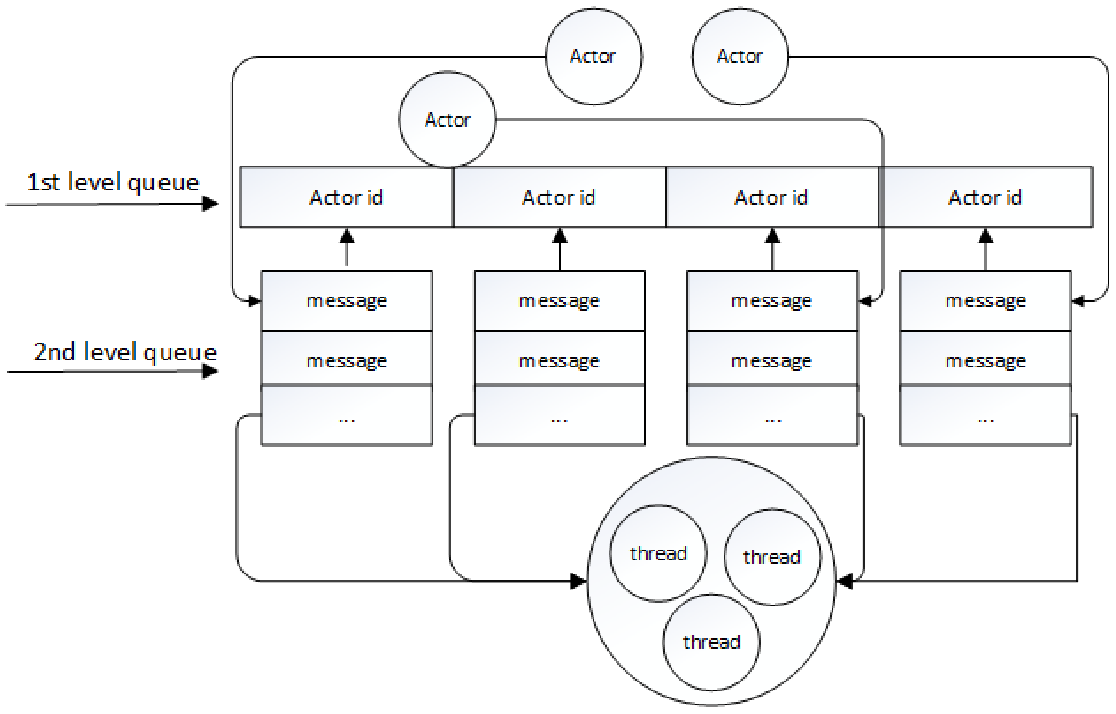
Figure 4.1. the abstract of system in actor model.



4.2 Message schedule

The scheduling of messages in the actor`s mailbox is synchronous and sequential. Furthermore, the scheduling of messages in the different mailboxs of actors is parallel because the messages in the different mailboxs of actors can be scheduled by the threads in the system in a synchronous way. In this section, it will introduce how the messages is scheduled by the threads in the system. The kernel design of the system is a two-level queue. The two-level queue can been in more detail in Figure 4.2.1.

Figure 4.2.1.the design of two-level queue



The first-level queue keep with the addresses of two-level queues that are queue of the message used by actors. When a actor sends message to the recipient, firstly, finding the postion in first-level queue of the recipient, secondly pushing the message to the queue of the recipient and waiting for scheduling of system.

When servers start, a work thread pool with some work threads will be created. The threads in the work thread pool loop to pop two-level message queue from global message queue and then pop one message from the two-level message queue, and then corresponding callback function will be invoked to deal the business of it.

Although, dealing with all mssage of the second message queue once is more efficent as which need less access of acotrs. But be fair of CPU`s scheduling, just popping one meesage from the sencond message queue every time is more reasonable, which make sure no actor use CPU all the times so that other actor be waiting all the times.

4.2.1 global two-level message queue

In the paper, we use pseudocode to describe the data strcut of sysytem. The pseudocode used in this paper is similar with Json: ‘[]’ is deemed as a queue or list,’ {}’ is deemed as a object or dict.

global_message_queue

```
global_message_queue = [
{
	"actor_id" = 456
```

```
        "server_message_queue" = [
        {
                source_id    = 123,
                session_id = 778899,
                data             = "xxxyyzzzz"
        }]
}]
```

From the struct of global_message_queue:

1.The server_message_queue is a queue of server_messages. The data name of the server_message:
source_id: actor id of sender
session_id: Some message sent to the recipient may need to respond. The session_id of message is used to match responding from the recipient. If the message need not to respond, session_id is set to 0.
data: message data sent to the recipient.

2.The global_message_queue is a queue of server_message_queue:
actor_id: the actor id of this server_message_queue belongs to.
server_message_queue: server_message_queue of the   actor_id

Compare-and-swap(CAS) for operations of queue
In our system, the global message queue is competitive resource as it is operated by multiple threads. Compare-and-swap is an atomic instruction used in multithreading to achieve synchronization in the system. This operation is used to implement synchronization primitives like semaphores and mutexes, as well as more sophisticated lock-free and wait-free algorithms. GCC[x] built in functions by using compare-and-swap for atomic memory access. There are some functions of it:

gcc

```
type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type__sync_lock_test_and_set (type *ptr, type value, ...)
type __sync_lock_release (type *ptr, type value, ...)
```

These built-in functions perform the operation suggested by the name, and returns the value that had previously been in memory.

lock and unlock in C

```
int lock;
#define LOCK(q) while (__sync_lock_test_and_set(&lock,1)) {}
#define UNLOCK(q) __sync_lock_release(&lock);
```

4.2.3 actor list

The management of actors is implemented using a list.

actor_list

```
actor_list = [{
	name,
	actor_id,
	message_queue_addr,
	callback_func
}]
```

for data struct of actor_list:
name: name of actor. actor id is dynamic when server start, but name is always same. When send a meesage to the actor, we usually use a name, while actor id is used in inherent server.
actor_id: actor id.
message_queue_addr: the address of message queue of own actor.
callback_func: when the message is popped, the callback funtion matching it will be invoked.

4.2.4 work thread pool and other thread

When the server start, a fixed number of thread pool wiil be started that depends on num of cumputer`s CPU(we usually set it N - 1(N is num of cpu)). When server is running, the thread pool loop pop message and invoke the callback function matching this message to deal with something.

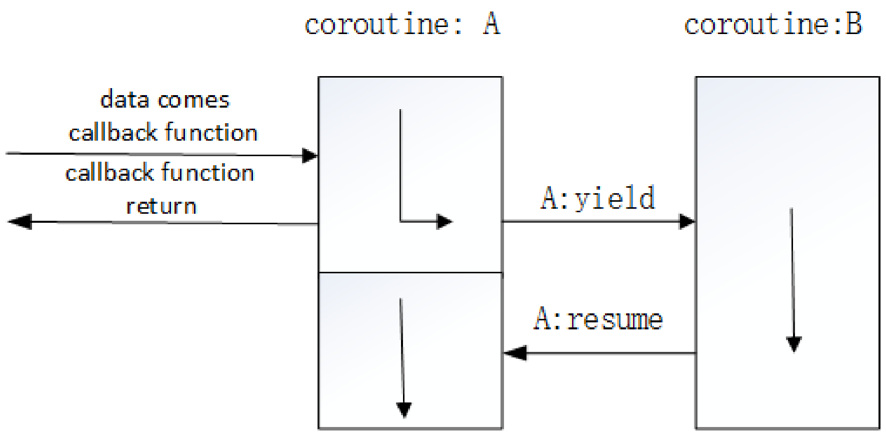
Moreover, when the server starts other theads includes timer thread used to deal spontaneously something, socket thread used to tackle with soket readable and writeable events.

4.2.5 coroutine in servers

Coroutines is used to guarantee sequential services. If we implement server use callback function. Coroutines is used to help us connect intermittent services that separate by callback fucntion in C/C++/Java. When work threads get a message, matching callback funtion would be invoked. If the message will be transpond to lua or python to deal, a coroutine would be created to deal with it.

While processing this message, if the coroutine send message(RPC) to others, coroutine.yield function will be invoked to make this coroutine suspend and wait for resuming. The callback function will return immediately, but lua or python will record this message session id and map the session id with the corotine. Then when it recevie respond message, coroutine.resume function will be invoked and coroutine will be resume and continue work. In this way, coroutines cooperate wth each others, which means that a task can be finished in coopearation by a group of coroutines. In this way, it guarantees sequential business logic. It can been in more detail in Figure 4.2.5.

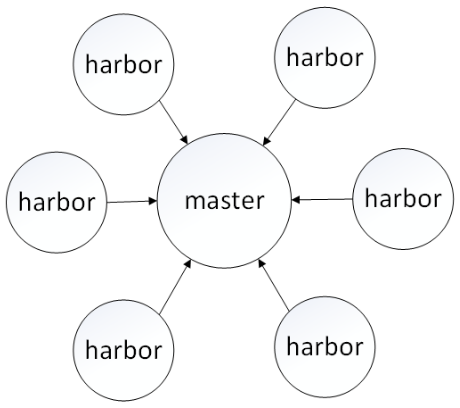
Figure 4.2.5. coroutine and callback function



4.2.6 cluster

In previous chapter, we introduce the design of a single node in the distributed system. After that, we will use master-harbor model to implement cluster of the distributed system. The cluster can been in more detail in Figure 4.2.6.

Figure 4.2.6. cluster



Master in the system
A master server as a center server to synchronize all actors data including name, id and address of the actors in cluster. It uses a key-value database in memory to implement a dictionary to store data of the actors.

master

the data struct of Master node

```
server_master{
      master id,
      server nodes address list,
      hashmap actor id and name
}
```

For the whole system, every actor id is unique. It is 32 bits that high 8 bits is computer id and low 24 bits is uesd to represent service id of the coumpter, which means that there are must 255 server nodes in cluster(but, it is ture that we can use more than high bit of actor id to support more server node if it is necessary). In this way, it is easy to distinguish between local address and remote address of actor.

So, the purpose of master node is to respond the query of actor`s name and address and synchronize actor data when some actor data updated. Byt the way, we normally advice creat one server node in one computer.

Harbor
The harbor save data of all server nodes in cluster, it is ued to communicate with others. When a server node start, it will connect to master server and then register its name and unique id to master. If successly, the master will synchronize data of this new nodein cluster. When the harbor receives the data of new node added, it will connect it to finish the structure of cluster.

harbor

```
server_harbor{
    actor id,
    master address,
    remote actors queue = [{
        remote actor id,
        remote actor name,
        remote actor addr,
        msg_queue = [{
            message data
        }]
    }]
}
```

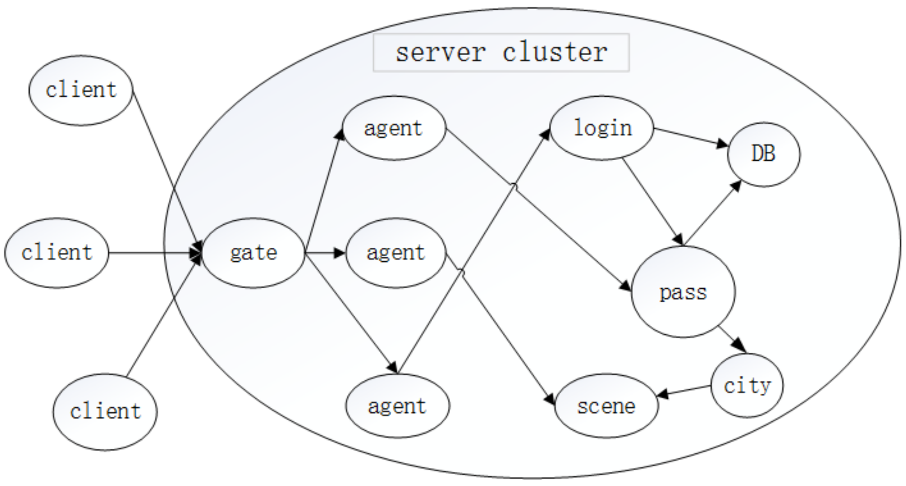
From the the data struct of harbor, we can see the server harbor save all data of nodes in the cluster. When sending a message, if this message is sent to remote server node. It will be pushed remote message queue in the harbor and wait for sending. When harbor receive a message from remote server node, the message will push its destination message queue and wait for scheduling.

4.3 gate and agent for client

The above mentioned is the inherent implement of the system, but it is necessary for a integrated system to connect and communicate with external clients or servers. For example, for a game server clients need to connect it and it need to connect database servers likely mysql server.

So, a gate service is necessary, which listens a tcp port and waits for connecting from clients. When it receives data from clients, it transpond data to agent service, and then agent service send data to inside of server to deal with it. Sometimes, the format of data in inherent services is different from the format of data from clients, the gate services can deal with it to meet requirement of inherent services. The system with gate and agent services can been in more detail in Figure 4.3.

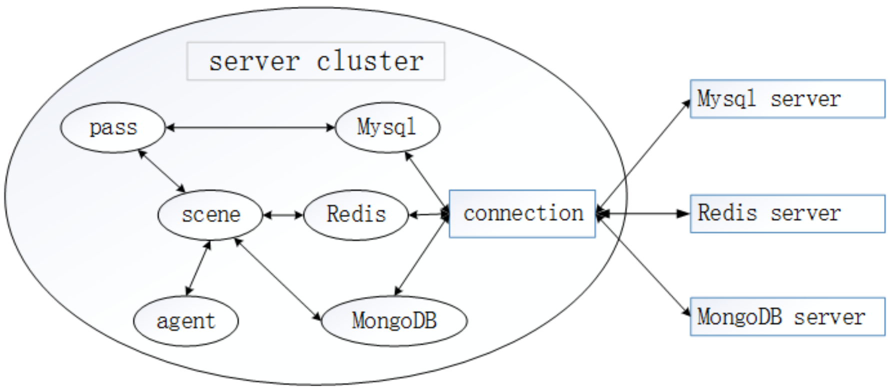
Figure 4.3. system with gate and agent services:



4.4 connection component

Another service is named as 'Connection'. It is different from Gate service, it is uesd to connect from server inherently to outside servers. The function of Connection is to listion different system fd readable and writeable status, which is implemented by epoll on linux. When Connection receive data from other server likely mysql server or redis server, it will trandspond data to inherent service. The system with connection component can been in more detail in Figure 4.4.

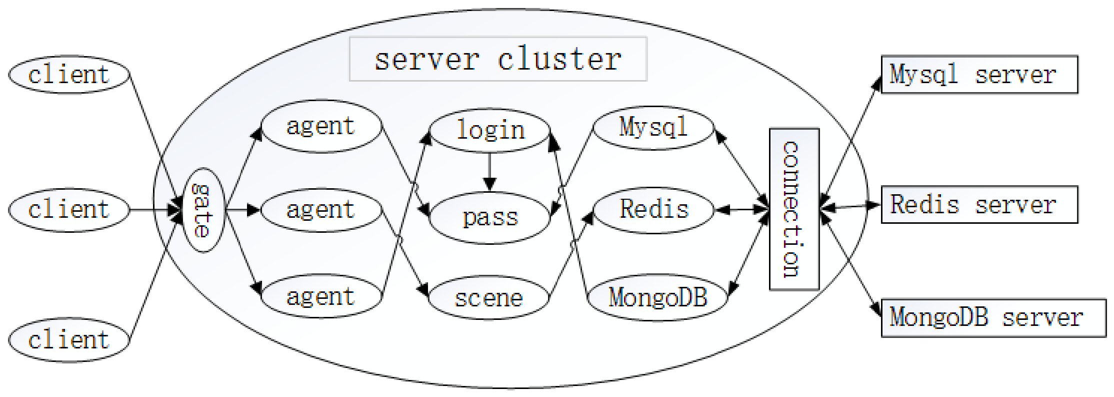
Figure 4.4. system with connection componet



4.5 summary

In this chapter, we designed a distributed server in actor model. In inherent system, services cooperate with each others by message-passing. Furthermore, the system can transact data with outside clients and servers using gate and connection components.The integrated system can be been in Figure 4.5.

Figure 4.5. integrated system



5. conclusion and future work

This paper presented a framework of the distributed server in actor model. It can be implemented in programming language C/C++ and lua, scala or other programming languages supported corountines. And It can be used in game services, web services or other systems.

In inherent system, the actors cooperate with each others by message-passing. The scheduling of message in the actor`s mailbox is synchronous and sequential, but it isn`t sequential all the time as the order of message scheduling in the same mailbox depends on the implementation of the mailbox, which implemented in FIFO or priority queue. But, the messages sent to different actors can be scheduled by the threads in concurrent or parallel way. In this way, the actor model is comprehensive of concurrency and parallel.

Reference

[x] multicore-crisis   http://blog.mischel.com/2008/08/04/multicore-crisis/
[x] Moore's law https://en.wikipedia.org/wiki/Moore%27s_law
[x] The free lunch is over http://www.gotw.ca/publications/concurrency-ddj.htm
[x] seven typical models of concurrent programming .Seven Concurrency Models in Seven Weeks: When Threads Unravel 1st Edition. by Paul Butcher
[x] The Lambda architecture http://lambda-architecture.net/
[x] MapReduce http://research.google.com/archive/mapreduce.html
[x] Mars Pathfinder
http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html
[x] Therac-25

http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=274940&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel1%2F2%2F6812%2F00274940.pdf%3Farnumber%3D274940
[x] multithreading concurrent paradigm. Why Events Are A Bad Idea
(for high-concurrency servers). Rob von Behren, Jeremy Condit and Eric Brewer
[x] Actor Model ACTORS: A Model of Concurrent Computation in Distributed Systems. Agha, Gul Abdulnabi. 1985-06-01
[x] Actor Model https://en.wikipedia.org/wiki/Actor_model
[x] Irene Greif (August 1975). "Semantics of Communicating Parallel Processes". EECS Doctoral Dissertation. MIT.
[x] Henry Baker; Carl Hewitt (August 1977). "Laws for Communicating Parallel Processes". IFIP.
[x] what is actor http://doc.akka.io/docs/akka/2.4.2/general/actor-systems.html
[x]mailbox http://doc.akka.io/docs/akka/2.4.2/general/actors.html
[x] gcc https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html