The Actor Model and Queues
Or
"Why Batch Processing is the New Black"

IEEE/ACM Princeton - ITPC 2017

Brad Whitehead, Chief Scientist – Formularity

March 17, 2017

I'm the Chief Scientist at Formularity. Formularity is a small company and you may not be familiar with us. We develop high security electronic enrollment forms for things like national identity management programs, financial institutions, and national health care program enrollments. We are dedicated to making sure the sensitive personal information you provide on our forms is secure and protected, even in a cloud environment. In addition to our forms being run by our clients in their data centers, we also offer a hosted solution. Since our target market are national scale enrollments, our hosted solution has a backend infrastructure that can enrollment millions of people per day. Actually, we can scale our backend systems to handle ANY number of enrollments per day. Specifically because we use the architecture I'm going to discuss with you today. My talk today is on the Actor model, Queues, and why Batch Processing is the New Black.

UIDAI

Unique Identification Authority of India
Planning Commission, Government of India

AADHAAR

Prior to helping to found Formularity, I was a Partner and Master Technology Architect at Accenture. My specialty, if you will, was national level biometric identification and border management systems.

Accenture was one of the three original Biometric Service Providers to the Government of India for their Aadhaar national biometric identification system. The Unique Identification Authority of India is enrolling 1.1 billion people in the Aadhaar program; all ten fingers, both irises, and face. We process 1 million enrollments per day and may move up to two million.

AADHAAR – The Last Day

Comparing 1 Million New Enrollees…

…Against 1 Billion Existing Enrollees

$1x10^6$ X $1x10^9$ = $1x10^{15}$

1 Quadrillion Comparisons in a day!

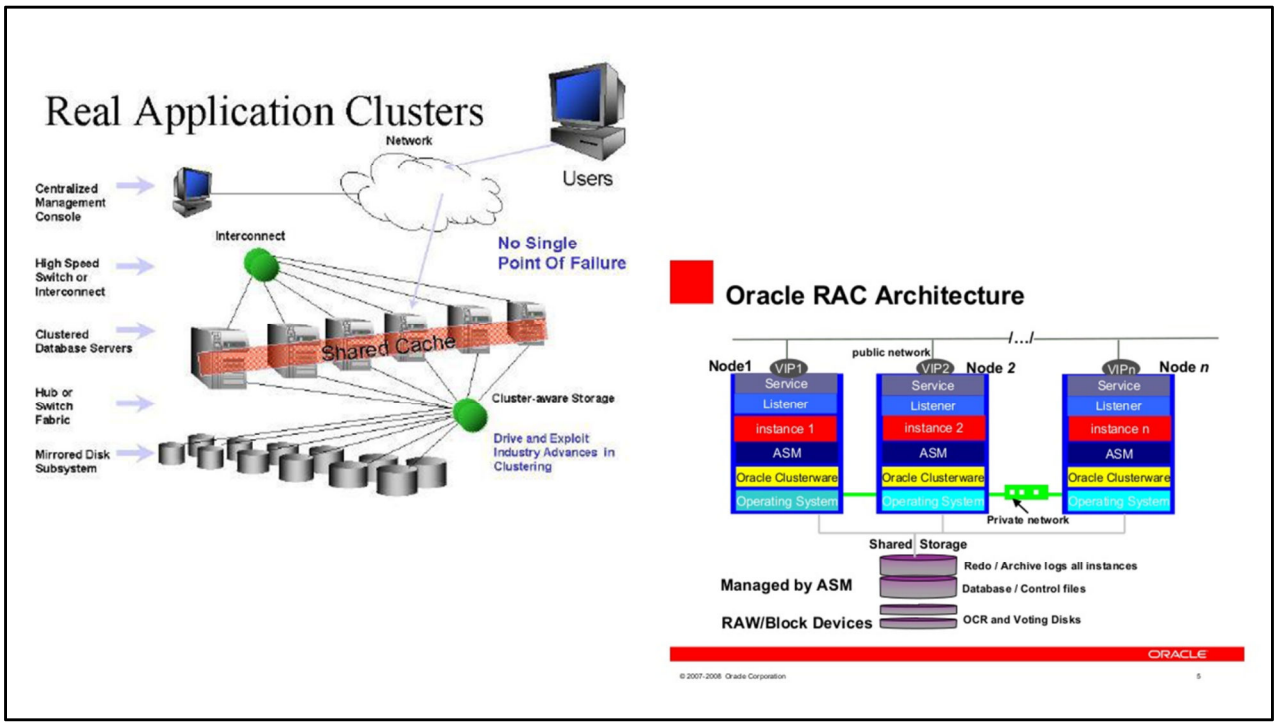
If you are using a database for workflow, that's:

11 Billion Transactions/Second!

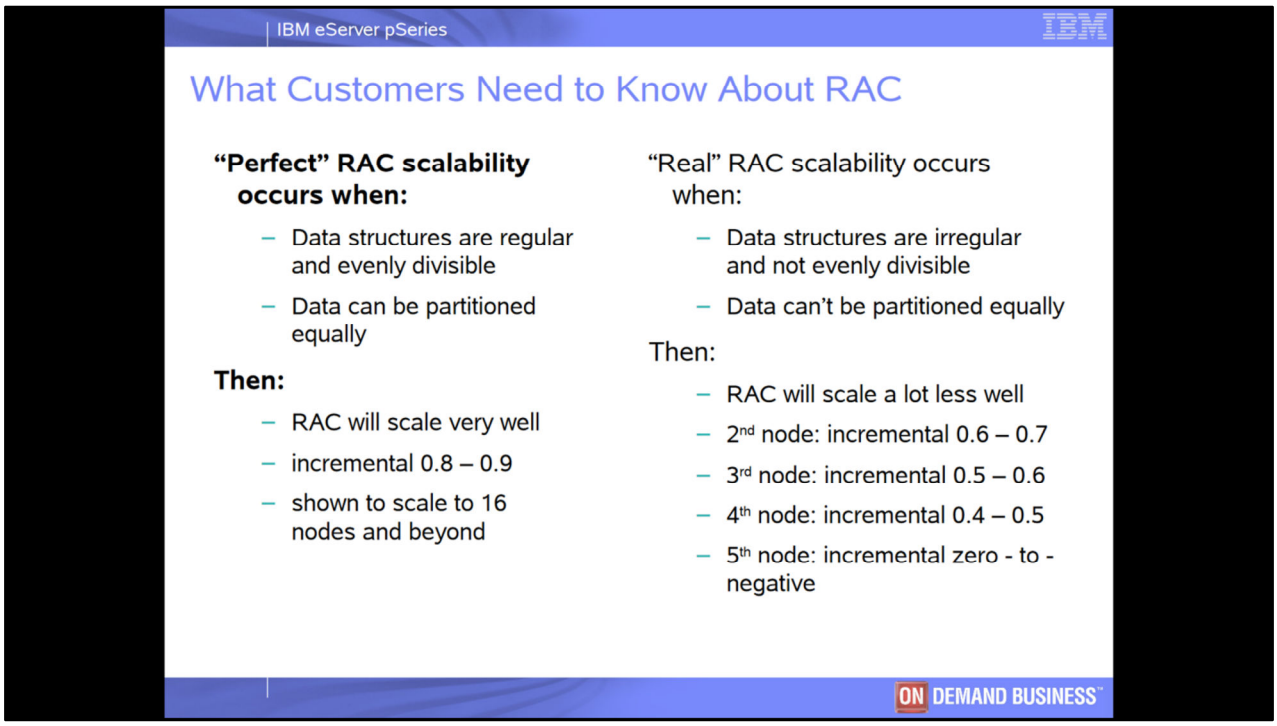
(It's Even More If You Individually Log Each of 10 Fingers, 2 Irises, and a Face)

I took over as Acccenture's Chief Architect on the program when we ran into scaling issues. Initially, we started by using the industry leading biometric middleware. This product uses an Oracle database as its workflow engine.

When we started processing over 500 thousand biometric enrollments per day, we hit the transaction limits of the Oracle system. And I don't mean the limits for our particular hardware infrastructure, I mean the absolute limits of the Oracle database. Oracle scales using an architecture they call Real Application Clusters.



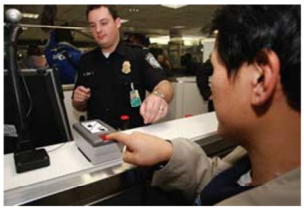
Basically, you have any number of Oracle databases running on separate servers, working together. Theoretically, you can have any number of servers in the cluster and it can scale to any workload. However, with real-world data loads, this infinite scaling doesn't happen. It's difficult to get good benchmarks and case studies on Oracle products because Oracle clients are prohibited by their license from publishing any type of benchmarking information,



…but one Oracle partner has published data that shows that for the average real-world workload, most Oracle Real Application Cluster systems stop scaling after 5 nodes in the cluster. This was certainly my experience in India. We had 4 nodes in the cluster and we couldn't push more transactions through the system by adding more nodes or by tuning the nodes we had. And when I say "we", I mean Oracle themselves. We had Oracle's own developers from both Redwood Shores and Bangalore helping us. My first job as Chief Architect was to replace the Oracle database-driven workflow system with a new workflow system using the methodologies I'll describe to you today. This workflow system easily handles 1 million enrollments per day and since we scale linearly, we can dial in how many enrollments by just increasing the number of servers.

US-VISIT

Keeping America's Doors Open and Our Nation Secure



U.S. Customs and Border Protection

While at Accenture, I was also the Chief Architect on the Department of Homeland Security's US VISIT program; now called the Office of Biometric Identity Management OBIM). This is the system used by Customs and Border Protection (CBP) at all ports of entry. When you enter the US at an airport or most land border checkpoints, you provide your fingerprints to the CBP agent. In 10 seconds or less, we check to make sure you are who you say you are and that you are not on the Terrorist checklist or the FBI's warrant system. This was the largest biometric identification system in the world, until we helped India build their Aadhaar system.

I was also Accenture's Chief Architect on DHS's Transportation Security Administration (TSA) Transportation Worker Identification Card system, or TWIC. Everybody that works at a shipping port or drives a hazardous cargo truck has to have a TSA-issued biometric TWIC card. We built the new TWIC system using the architect I'm talking about today.

So, I have the real-world experience on scaling large national level critical systems, using the architects I'll be talking about today

I'm also old!! It's good to be old!!!

"…what has been will be again,
what has been done will be done again;
there is *nothing new under the sun"*

- Ecclesiastes 1:9, Hebrew Bible

Everything old become new and exciting again

"Those who cannot remember the past are *condemned to repeat it*

– George Santayana, 1905

In other words – Been There, Done That, Have the (moth-eaten) T-Shirt!

This is never more true than in Computer Science and Information Technology. For a “young” discipline, it’s amazing how many repeat cycles we have seen.

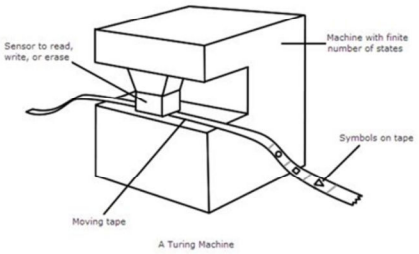
Let's start at the beginning, or at least 1936…

Who knows who this is? Alan Turing is famous for many things. Being persecuted by the British government…for cracking the German Enigma machines during World War 2… for developing the Turing Test used in Artificial Intelligence research. What I'd like to discuss today is his "Turing Machine" thought experiment.

1936AD

Turing Machine

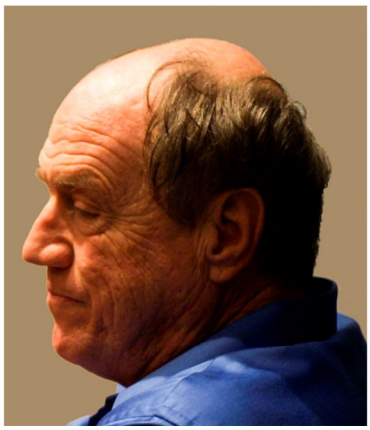
- a theoretical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.



A Turing Machine

I assume that most of you are familiar with the Turing Machine, so I won't spend a lot of time on it. The Turing Machine is a finite state automation that can model any computing process. Its composition is extremely simple, consisting of an infinitely long tape divided into cells; a fixed set of characters that can be individually written into the cells; a read/write head positioned over the tape that can read or write a character in the cell; a mechanism to move the tape forward or backwards under the head one cell at a time, and finally, a look-up table of machine states and actions. This last part is the "program" of the Turing Machine. The Turing Machine controller…and Turing didn't necessary think that the controller was a machine. His notes on what he called the A-Machine makes references to a possibly human controller. The controller looks at the current state of the machine, i.e., the character being read and the last matched row of the lookup table and looks up this new state in the table. The newly matched row tells the controller which way to move the tape and what character to write in the resulting cell. From these basic steps, you can write any solvable computation as rows in the state table. Programming a Turing Machine is long and difficult and actually running useful program would not be very efficient.

More efficient than giving a hundred monkeys type writers but probably only barely so. Still, the Turing Machine is a small simple algorithm that can be used to model any modern non-quantum computing method. Think of it like the Lego block of Computer Science. One of the things you'll note is that a Turing Machine is single threaded…



Carl Hewitt

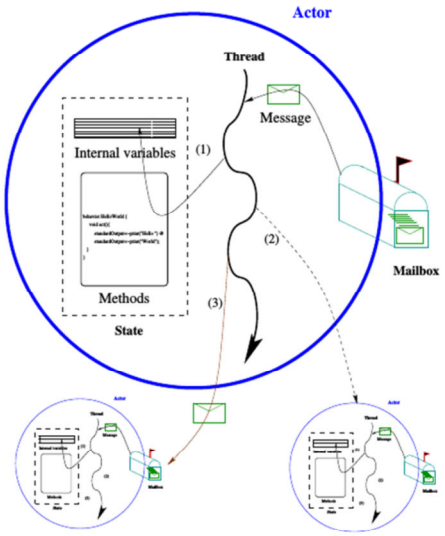


Figure 1.2: In response to a message, an actor can: (1) modify its local state, (2) create new actors, and /or (3) send messages to acquaintances.

The Actor Model

In 1973, Carl Hewitt refined the single threaded Turing Machine into a model of concurrency he labeled as the Actor Model. An Actor is an automation that receives external input through messages. This is somewhat analogous to Turing's infinite tape. Based on a received message, an Actor automation can do one of four things, it can:

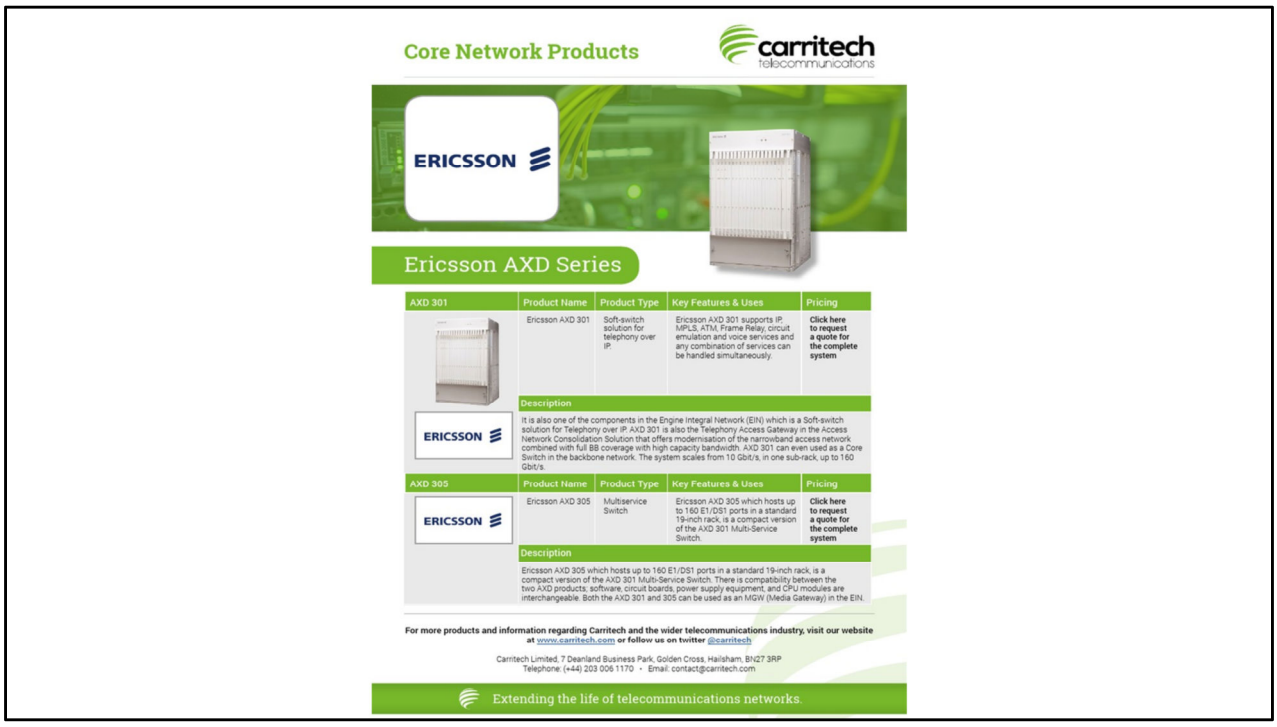
1) Make local decisions based on its finite state table (analogous to the Turing Machine Controller);

2) Create new Actors (since Turing's machine was a single instance, there is no equivalency in the Turing Machine and this forms the basis of Hewitt's concurrency attribute);

3) Send out messages (akin to writing to Turing's tape); and

4) Adjust its internal state to a new state (which will reflect on how it acts to future messages)

Again, simple…..but unlike the Turing Machine, practical instances of Actors can and have been written. For example, the programming language Erlang is Actor-based.



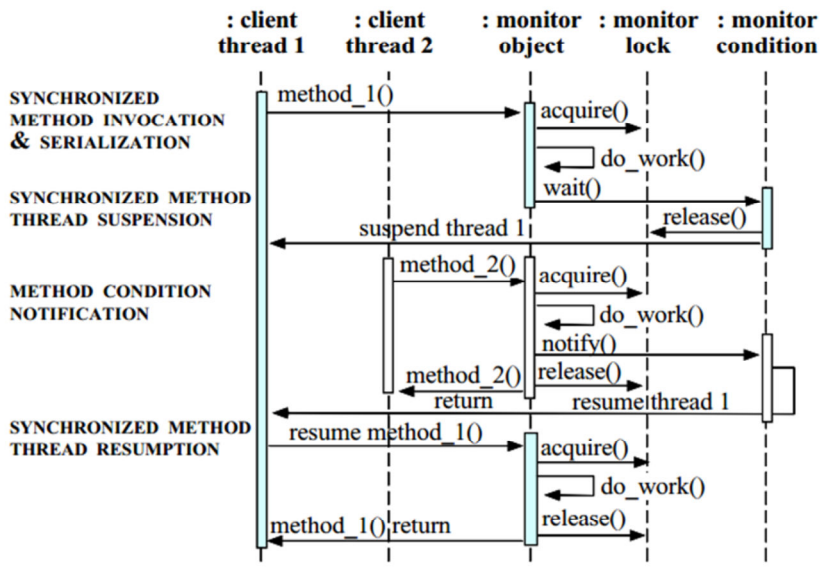
Designed to operate the thousands of simultaneous conversations in a telephone switch, Erlang and the Actor Model are directly responsible for the Erisson AXD310 telephone having a stated high availability of 9 “9s”. That’s a down time of 31 milliseconds per year. That’s better than anything I’ve ever achieved in a Tier IV Data Center.

The Root of the Problem: Shared Resources

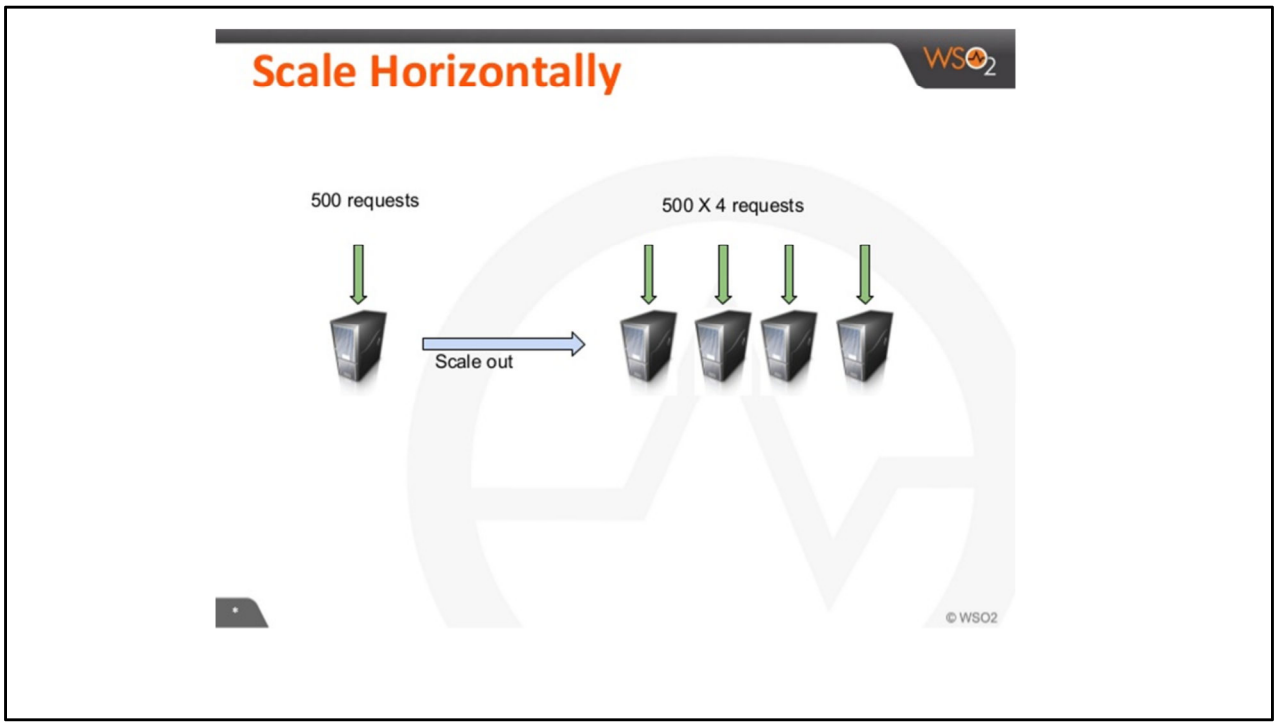
What Goes Wrong:
- Corruption
- Race Conditions
- Deadly Embrace
- Blocked Processes and Threads

Solutions(?):
- Locks
- Mutexes
- Semaphores
- Latches
- Monitors

Let's stop and talk about the concept of currency for a moment. Early mainframes had one processor or means of executing. So it did everything really fast and any illusions of concurrency were just that – illusions. Everything was sequential. So everybody built faster processors. The next step up in power was to add multiple processors to the mainframe. Four was generally the limit because they had to share memory and I/O connections. These multi-processors worked well so long as each processor was doing a separate, independent task. One processor might be calculating employee pay while another was handling customer service reps taking product orders. However, as soon as more than one processor started sharing the same resource, problems began to emerge. Unless the processors were physically synchronized, they could change each other's intermediate results. And if you physically synced them, you artificially restricted their speed. So we developed resource sharing mechanisms like locks and semaphores. Programmers had to start worrying about things like race conditions and "deadly embraces." And if they didn't worry about them, errors and failures occurred.

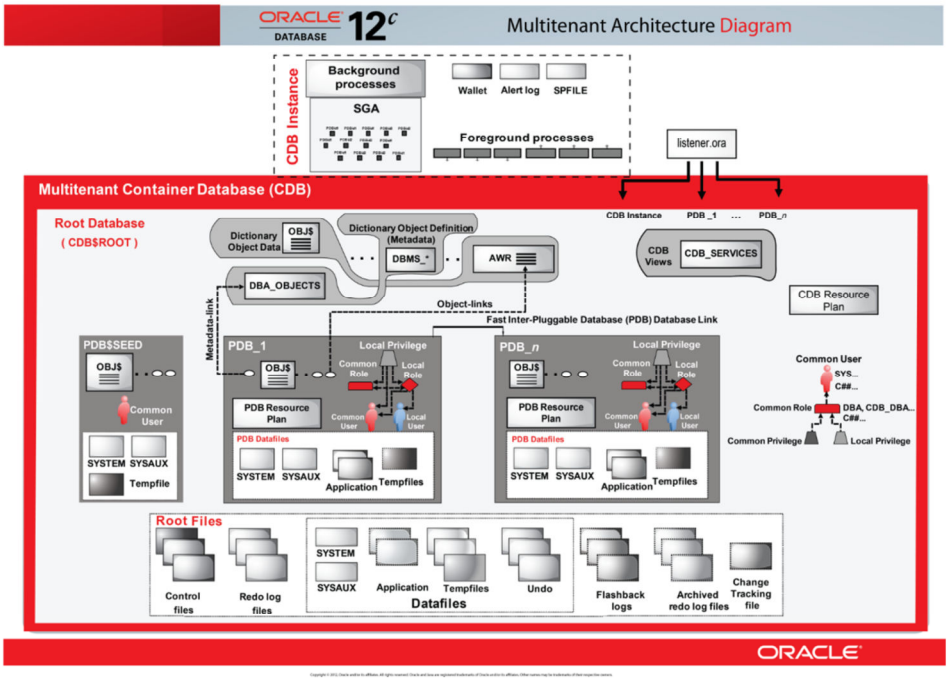


The same thing has happened on modern microcomputers. We complied with Moore's Law by making our processors faster until we hit physical limitations. Then we developed multi-core microprocessors….and hyper-threaded multi-cores. We took advantage of these multiple cores and hyper-threads through concurrent processing; breaking our monolithic process into multiple, interdependent programs or multi-threaded programs. And we got all the problems of shared resources; locks, semaphores, race conditions, and deadly embrace.



So, now we have exhausted the scaling capabilities of a single computer. We are clocking the processor as fast as we can, and we are using all its cores and hyper-threads. How do we scale? Horizontally. We add multiple computers clustered together to increase the parallelism and apparent speed and scale of the application.

Finally, we are on the right track. Independent computers offer more resiliency and availability.



However, so long as all the computers in the cluster are inter-dependent, we still have all the drawbacks of locks, race conditions, etc. We have to use locks and mutexes; which slowdown throughput and are the source of errors.

This is where Carl Hewitt's Actor Model comes into the picture. By decomposing our complex monolithic program into a set of simple services, we can take advantage of the inherent concurrency of the Actor Model.

So, what are these advantages?

1) Each Actor or step in the process is small, accomplishing a single function. The finite state table (or its programming equivalent) is bounded and traceable. This means that the "correctness" of each Actor type is more easily established

2) Each Actor type can be better tested because of its encapsulation nature and message passing framework

3) Actors can be easily improved or re-factored without affecting the rest of the application

4) Actors can be easily connected together to respond to changes in requirements or to provide new services

5) The only way one Actor can affect another Actor is through explicit messaging. This makes it much easier to see and avoid concurrency problems.

6) Actors can create new Actors, to replace failed Actors or to increase the processing power at any step in the overall process

7) Actors can encapsulate common resources and act as service or resource provider

8) Multiple Actors can increase availability and/or persistence

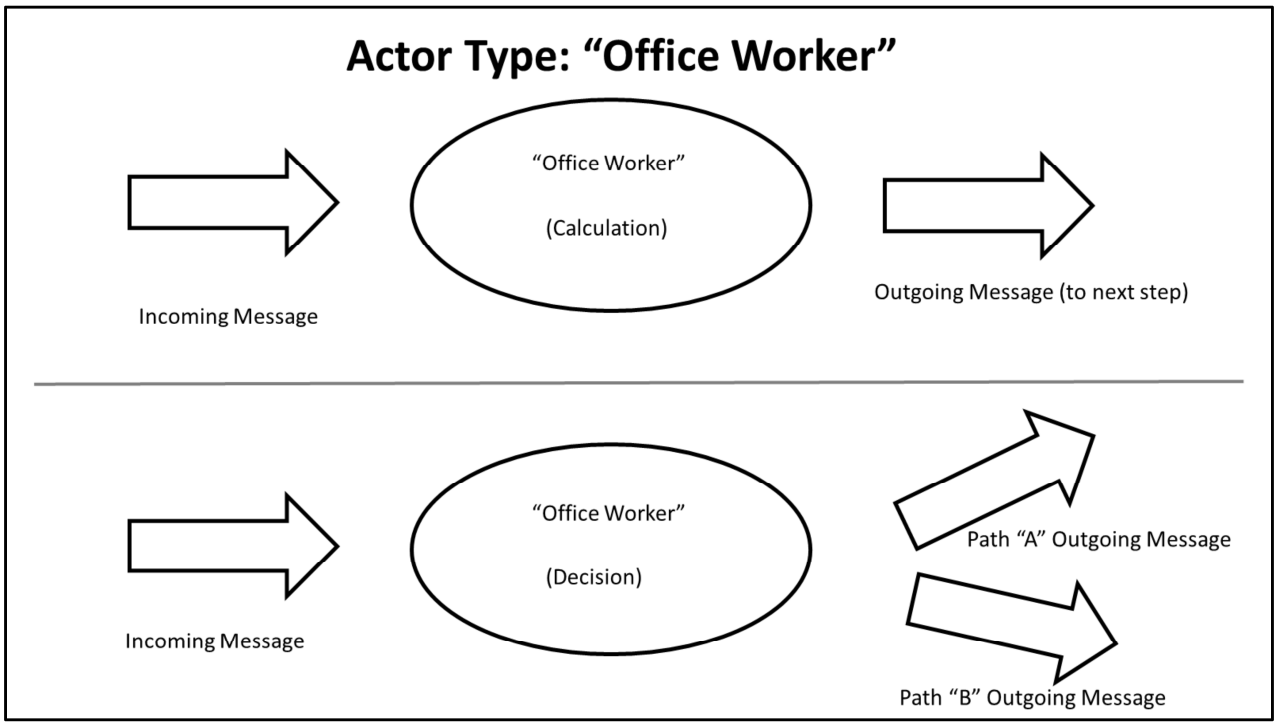
9) If each Actor is run on a separate server, the Actor code may fit entirely within the processor cache, greatly accelerating the execution speed

10) If each Actor is run on a separate server, geographical diversity for improved resiliency is possible.

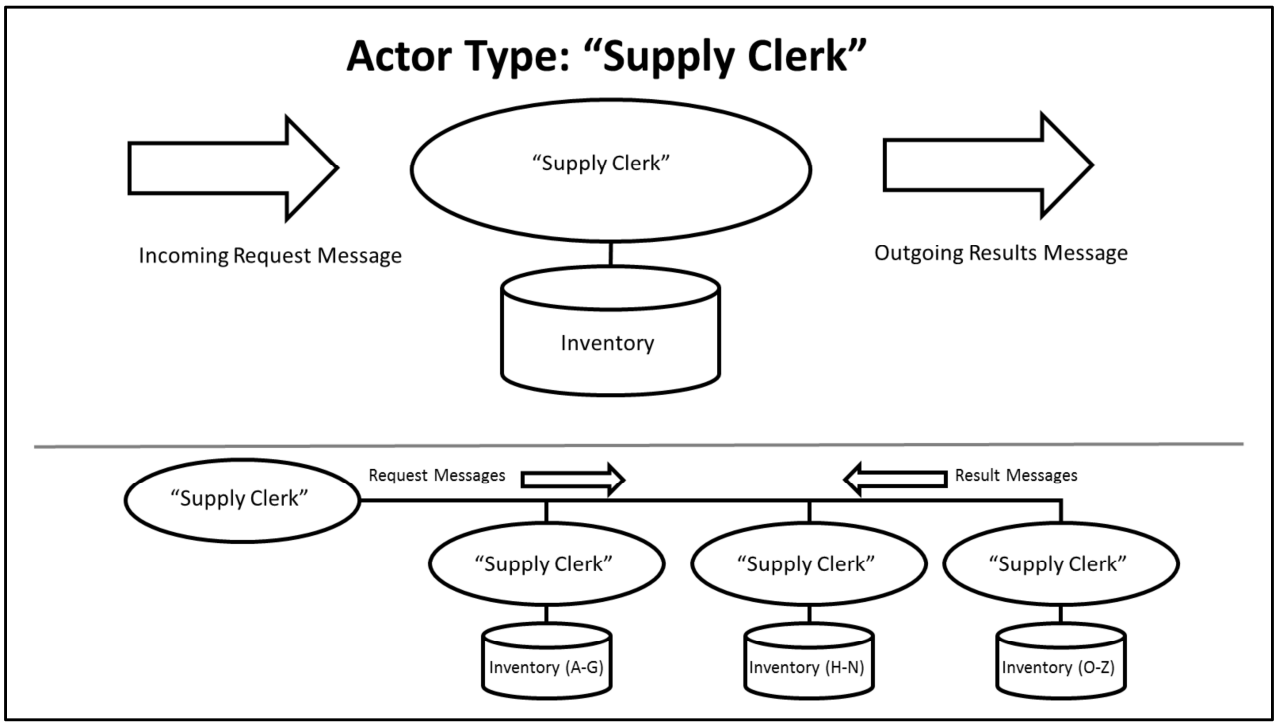
Actors = Microservices

If decomposing complex monolithic programs into smaller, concurrent chunks sounds familiar, well it is – The hot new buzzword today is “microservices”. If you don’t know about the Actor Model or don’t remember it from school, you know microservices are small bits of functionality, but you don’t have any guidelines or frameworks. By using our “old” knowledge and experience of Actors, we now have a proven model and pseudo-template to use in defining our microservices and how they interact. No need to re-invent the wheel ;-)

It has been my personal experience over the last 40 years that human nature quite often provides the best way of doing something. So my first step is always to identify how a human would accomplish a given task and then see how it can be applied to an automated system. This isn’t foolproof, but it’s a good starting point. So, if we consider our Actors to be actual humans, what services are necessary and how can we implement them?

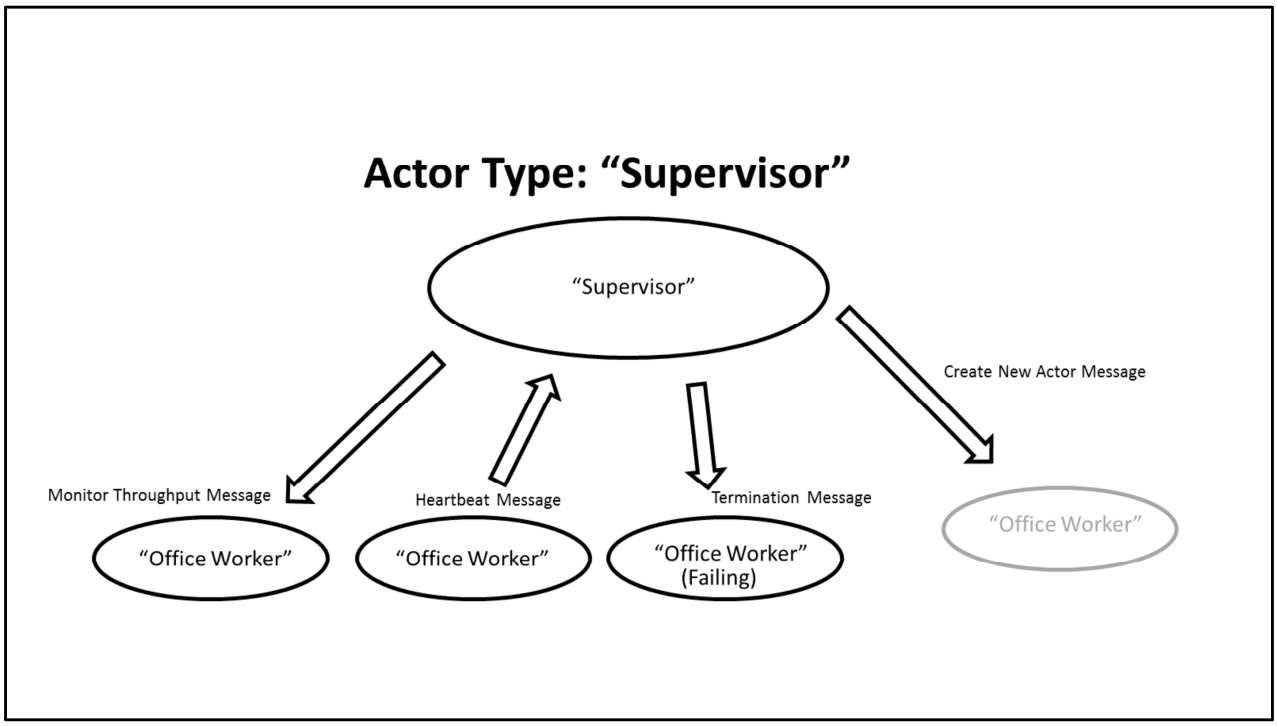


Our first Actor type is the “Office Worker”. The Office Worker takes messages out of its message queue, performs an action on them and then sends out the results as new messages. It’s analogous to an office worker at his or her desk. The process may be linear, such as performing a mathematical calculation; or it may involve a decision. If it’s linear, then the Office Worker-type Actor will probably send one message out to the next cluster of Actors in the overall process. If the Office Worker performs a decision, the resulting message may be addressed to different clusters in the application.

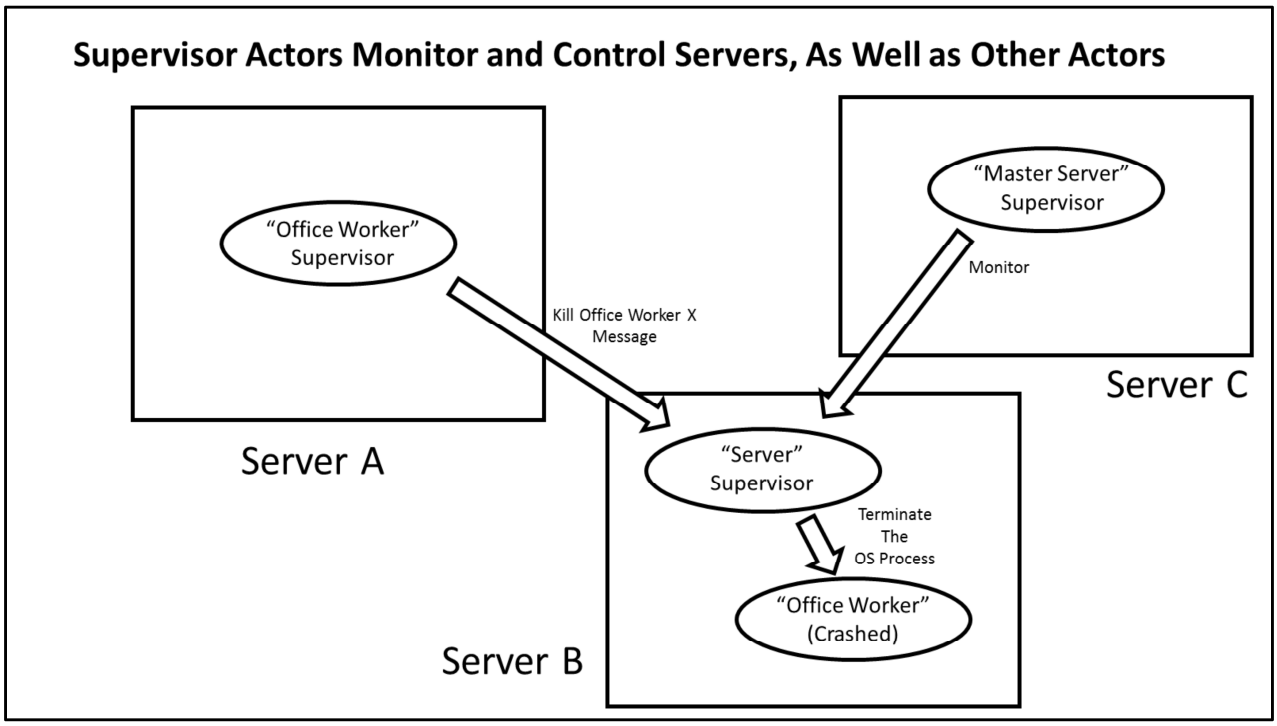


Our next Actor is the Supply Clerk. Remember Radar O’Reilly from the movie and television series M*A*S*H? Radar was responsible for all things administrative. Nobody else could find or do anything administrative but Radar always took care of everything and never (well, hardly ever) made a mistake. The Supply Clerk-type Actor encapsulates common resources. Need to update inventory? Don’t allow any old Actor to change the inventory. If you do, two or more Actors will make conflicting changes. Make the inventory the responsibility of a single Supply Clerk Actor instance. If the inventory needs to be updated, send the update request to the Supply Clerk Actor. Problem solved. So what if you have too many inventory changes per second for one Supply Clerk? Well, Radar often coordinated with other Supply Clerks to get a job done. Our computing equivalent are the “scatter-gather” or the “map-reduce” algorithms. We may have multiple levels of Supply Clerk-type Actors. The primary inventory manager Supply Clerk may send our update message to multiple subordinate inventory Supply Clerks, each responsible for its own subset of the inventory. Or we may use multiple inventory Supply Clerks for redundancy and consensus. Two factor transactions are ACID but they are also expensive from a processing time standpoint. For that matter, large portions of the Internet insist that “eventually consistent” is sufficient. However, if we use redundancy and

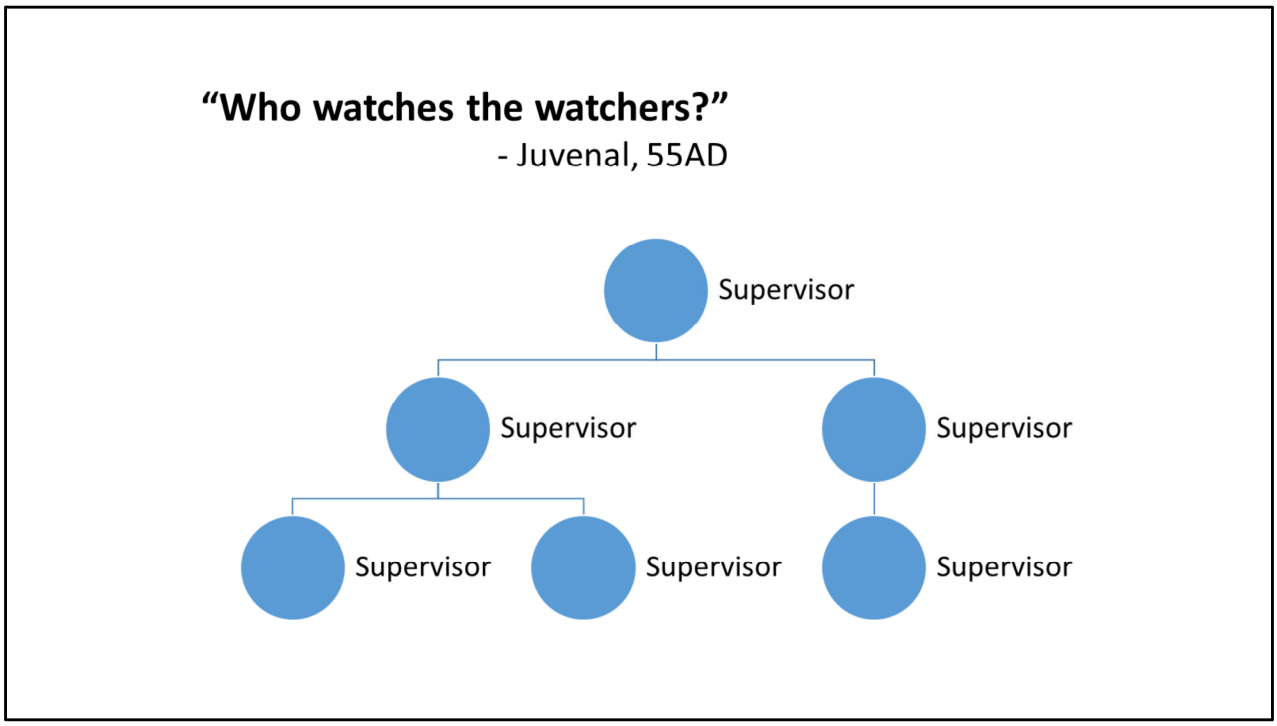
consensus, we can end up with nearly transactional “strength” within “eventually consistent” timeframes. As the architect, you have to make the best decision based on the requirements. The flexibility of the Actor Model doesn’t lock you out of any options.



Our third Actor type is the Supervisor. We all know the Supervisor. Doesn't do any useful work but manages the labors of the producers ;-) This doesn't change in the Actor Model. But the Supervisor is responsible for the overall application running smoothly and meeting its processing time requirements. The Supervisor monitors the health of all the Office Workers and Supply Clerks under its supervision. It can do this by observing the throughputs of the subordinate Actors, or it can actively ping subordinate Actors, or by receiving "heartbeat" messages from subordinates. Regardless of how it monitors the other Actors, it is responsible for messaging failing Actors to gracefully terminate themselves (after they complete their current process cycle), or killing the operating system process of the failed Actor. The Supervisor Actor monitors the performance of the processing cluster for which it's responsible and for 2) starting new Actors to handle increased workloads, or 2) messaging surplus Actors to terminate when the workload decreases. How it does this workload monitoring is critical and one of the major "secrets" to designing an infinitely scalable web service. I'll talk about that next. However, just to finish up on the Supervisor Actor, Supervisor Actors definitely are hierarchical in nature and must cooperate with each other. Let's use two practical examples.

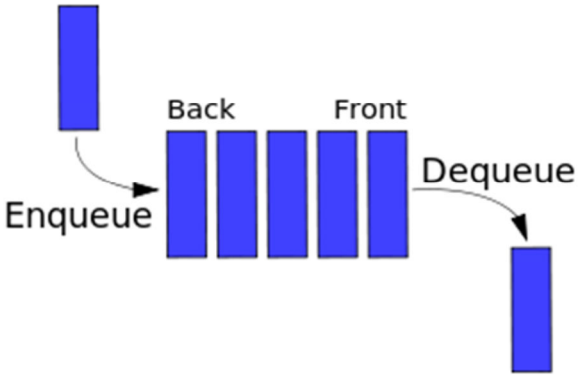


First, I mentioned the Supervisor asking a failing Office Worker-type Actor to terminate itself. This only works if the Office Worker Actor is still interacting. What if it's stalled in a loop and won't read the termination message? The Supervisor Actor has to ask that the operating system process housing the failing Actor be terminated. If the Supervisor Actor is on a different server than the Processor Actor, how is this done? Well, you need a set of Supervisors responsible for making sure the servers themselves are running. So, in addition to a Supervisor managing a set of Office Worker Actors, you have a Supervisor managing the servers. Our Office Worker Supervisor asks the Server Supervisor to terminate the failed Office Worker Actor. The Server Supervisor sends the message to the Actor on the server responsible for managing that server. If it kills the failing Office Worker great. If it doesn't, then it's also failing and the Server Supervisor is responsible for killing the whole server and restarting a new instance.

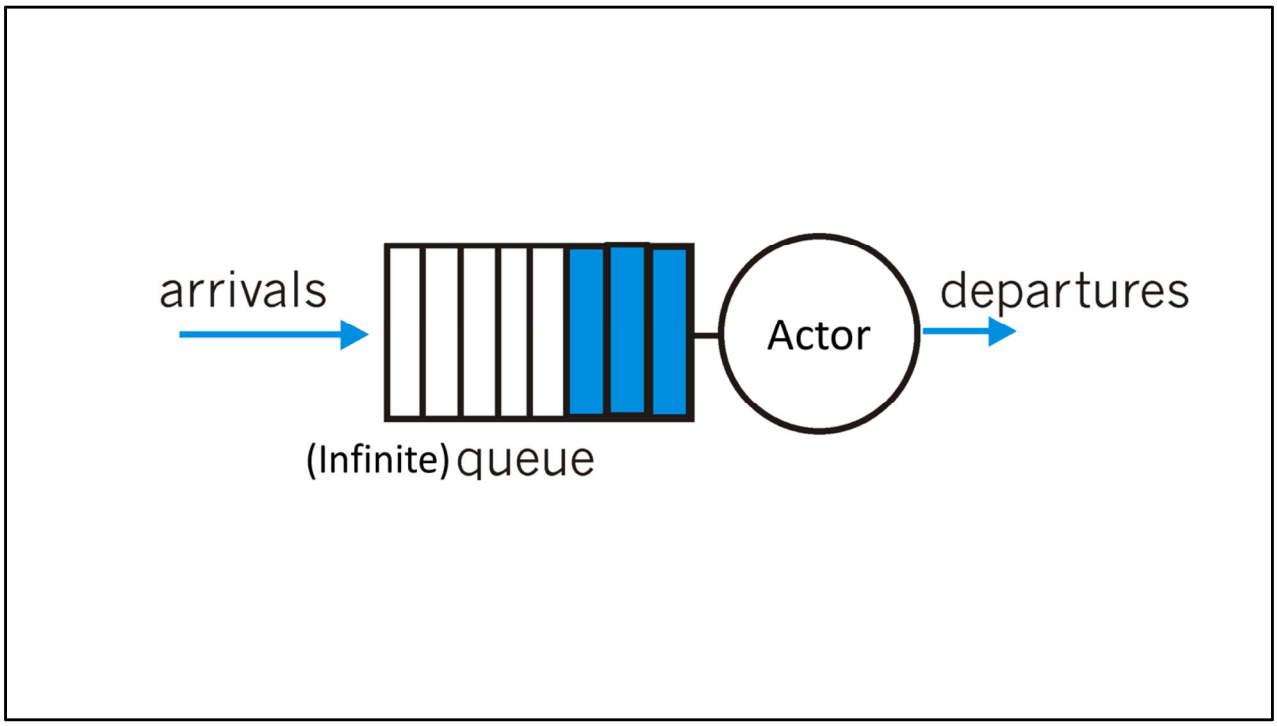


The second example of Supervisor hierarchical cooperation is in supervising the Supervisors. Or to quote the Roman poet Juvenal "Who watches the watchers?" Supervisor Actors have their own Supervisors. At the very top layer of our web service or application, we have a human operator, tied to a pager (what's a pager???? Anybody seen one recently?), getting "All quiet" messages on a periodic basis.

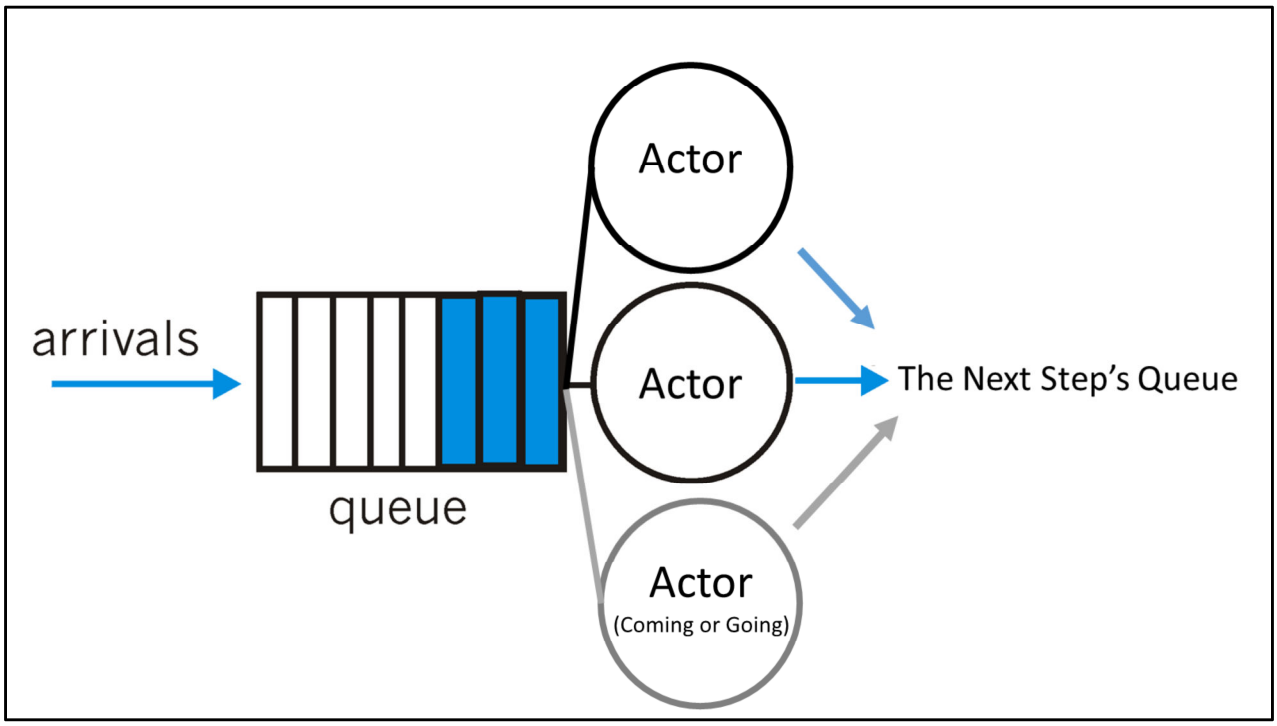
QUEUE



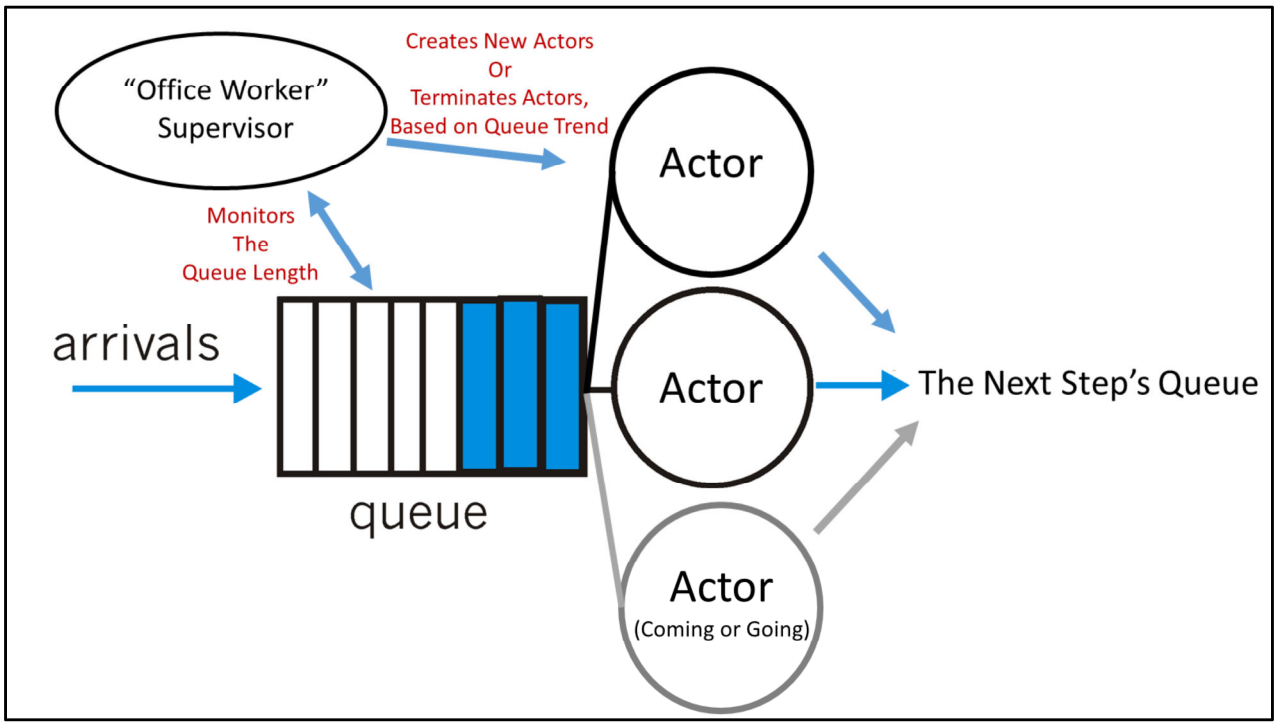
OK, now on to the second ancient piece of Computer Science technology that's key to our robust, reliable, and infinitely scalable cloud application. The queue is even older than the Actor Model. Coincidentally enough, one of the first scientific papers published on queuing theory was by A.K. Erlang in 1909, the same Erlang for which the Actor-based programming language is named.



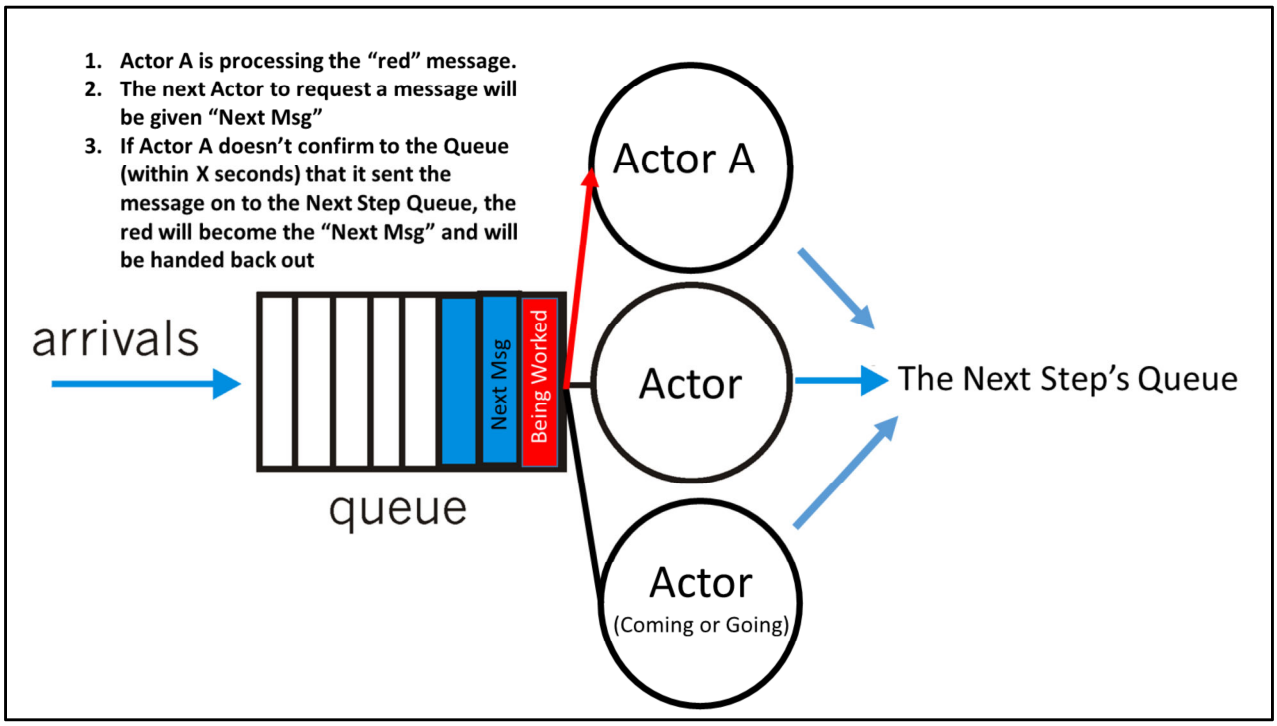
The best way to connect our Actors is through queues, rather than direct connections. Why is this? Again, let's looks at the advantages. First, it gives the whole system flexibility. If an Actor or set of Actors can't process messages fast enough, then the messages build up in an essentially infinite queue instead of being lost. Momentary slow-downs in processing because of things like garbage collection or virtual machine startup are automatically buffered by the interconnecting message queues.



Second, I have talked about clusters of Actors. If one instance of an Actor is insufficient to handle the workload, multiple instances can be created, each picking the next piece of work to be done off the queue. With the queue, instead of individual mailboxes for each Actor, Actors can come and go and they all share in processing the workload. So an Actor won't send a message off to another Actor instance. Instead, it will send messages off to the queue(s) of the next Actor cluster.



So far, this is analogous to a picking list in a warehouse and a staff of pickers. Each picker takes the top request off the picking list and goes into the warehouse to retrieve the required item. How do you determine if you have enough warehouse staff? Simple, watch the pick list. If it gets progressively longer, you don't have enough staff. The pick list will continue to accumulate requests until you can hire enough staff to stabilize or reduce the list. Simple. Nothing lost and you were able to monitor a gradually changing situation. What does it mean when the pick list is empty and you have warehouse staff standing around drinking coffee? Time to redeploy them to another part of the company. With our Actors, we'll just ask them to terminate themselves. OK, let's connect a couple of dots here. I said the Supervisor Actor monitored the workload and started or stopped Office Worker or Supply Clerk Actors based on this workload. The Supervisor monitors the workload by watching the input message queue to the Actor cluster. The Supervisor sees the queue increasing in size long before the situation becomes critical. If you monitor workload by measuring the CPU utilization of the server, or the I/O throughput, or the memory utilization, you are going to see sudden spikes that may kill your server or cause your Actors to fail before you can react. The queue gives you a much better and safer means of monitoring.



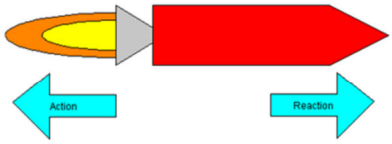
What happens if an Actor dies before completing the processing of a message? Is the message lost? [What do you mean you lost my deposit!!!!?] Or do you devise a scheme where messages have to be retained by their originating Actor until the processing Actor can confirm completion? With this type of complexity, you are setting yourself up for failure. Instead, "use the queue, Luke". When an Actor instance takes a message out of the queue, don't destroy the message, just put it in a "being worked" status. If the processing Actor doesn't tell the queue it has completed the requested action within a certain timeframe, the queue can just move the message back into the "ready" status and give it to the next member of the cluster that asks for a message. The message isn't removed completely from the queue until it's been completed and acknowledged. Worried about a whole queue failing? Simple – send all the messages to two redundant queues. Let the two queues keep themselves in relative synchronization. If an Actor finds it can't talk to its primary message queue, it just connects to the redundant secondary queue and keeps on processing. You won't lose any messages but you might re-process several messages. If the processing isn't idempotent, then create a unique key for each message and keep track of keys. Discard any duplicate messages or results. This avoids [What do you mean you deducted my one withdrawal twice!!!!]

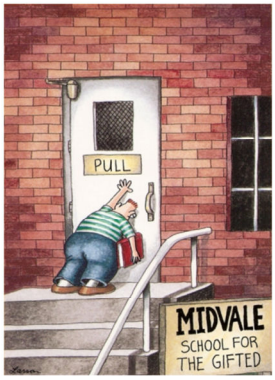
- Use Public Key cryptography to authenticate connection requests from Actors to Queues
- Use Cryptographic Hashes (digital signatures) to both confirm the originator of Messages, and the integrity of the Messages
- Use Encryption to protect the contents of Messages

Cryptography is Cheap!!

Use it Everywhere!!

Finally, queues eliminate active load balancers and the need to register and deregister Actors. As Supervisors create new Actors, the Actors start pulling the top message out of the queue. We have already discussed what happens if an Actor dies or terminates. Again, no need to deregister. If you have security concerns [and you should!!!], use public keys to authenticate connection requests to a queue, use cryptographic hashes (digital signatures) to confirm both the originator of a message and the integrity of a message, and use cryptographic encryption to protect the contents of a message. Cryptography is cheap. Use it everywhere!





We'll be saying a big hello to all intelligent lifeforms everywhere and to everyone else out there, the secret is to ~~bang the rocks together~~ pull, don't push, guys."

- Douglas Adams, Hitchhiker's Guide to the Galaxy

Fudd's First Law of Opposition: "If you push something hard enough, it WILL fall over."

- Firesign Theater, 1971

Load balancers! Before I tell you why load balancers are evil, let me talk about the third critical item for a successful large web application. Philosophy and Newton tell us that there are always two forces in equal balance. Call them Yin and Yang, or Action and Reaction. Or Push and Pull. In data flow, you can push data or you can pull it. And I'm here to tell you unequivocally that most system architects get it wrong! In Douglas Adam's "The Hitchhiker's Guide to the Galaxy", the quote is "We'll be saying a big hello to all intelligent lifeforms everywhere and to everyone else out there, the secret is to bang the rocks together, guys." To paraphrase Mr. Adams, the secret to scalable processing systems is really to "pull", not "push". But you knew that. After all, your parents and your teachers always told you not to push. Why not? Well to quote another old comedy sketch [anybody here old enough to remember Firesign Theater?], there's Fudd's First Law of Opposition: "If you push something hard enough, it WILL fall over". When you push messages or requests to a server, you end up crashing the server. If you are monitoring the CPU or memory of the server, you'll get a warning before the server crashes, but will it be in time to allow you to start up new servers and to register them with your message pusher? We've already seen queues give us sufficient time to start new Actors and there's no need to register with a message pusher when the Actors are

responsible for pulling their own messages.

If you don't believe me, here's Lucille Ball to demonstrate the flaw of pushing work. Do you want to do this to your own processes?
<https://www.youtube.com/watch?v=8NPzLBSBzPI>

Actors are computer programs and as such they aren't lazy. An Actor will process messages as fast as its execution environment permits. OK, to be fair, let me clarify – you can push messages to queues. After all, they are just glorified accordion pipes. They can accept and store messages much faster than your Actors can process them. And if a single queue (or redundant queue pair) can't keep up with the incoming messages, create multiple queues and let your Actor cluster pull messages from the multiple queues in a "round-robin" fashion.

To finish queuing off, load balancers are a symptom, not a solution. Load balancers are subject to configuration problems, especially as you are adding and removing Actors from a cluster. They push rather than allowing processors to pull. How do they know which Actor to push to? Round robin? That has the potential to leave processors idle. By monitoring CPU loads? That's not an exact science. And what happens when an application does a garbage collection pause after your load

balancer pushed a message to it? Finally, what happens when your load balancer fails? Sure redundant load balancers. But now you have to have coordinated communications between the two load balancers and their network connection has to be reliable.

So, the keys to building a successful, infinitely scaling web application is to use the 1973 Actor Model now called a microservice; the 1909 Queue, which is just batch programming; and being polite and not pushing.

Maintaining State

The key to being able to create new instances of Actors to handle the workload is that they must be "stateless"

- Each new message is a new action, without any dependence on previous messages or actions

Don't

- Don't store processing state in a database or shared memory
  - A database adds significant complexity to the resiliency and reliability of the application
  - Databases are slow
- Don't use shared memory
  - This becomes a shared resource that must be protected (with locks, mutexes, etc.)
  - Shared memory doesn't scale

Do

- Keep all state information in the message
- When an Actor reads a message, it should have everything it needs to perform it's task
- This implies that messages are not immutable

I just want to touch on two remaining aspects of a successful large scale program; 1) how to maintain state and; 2) how to handle long-running processes.

I've talked about Actor clusters and multiple Actors pulling their work assignments out of a common queue. This only works if the Actors are, for the most part, "stateless". Each Actor sees each new message as a new task, with no knowledge of previous tasks. I could tell you all the ways that traditional web servers keep or share state, but I don't want to drag this out. In an Actor model system, put the state in the message. In other words, when an Actor puts up a new message, the message holds all the information required to accomplish the Actor's process. The Actor doesn't have to go fetch the process state from a data base or from shared memory. This also implies that messages are not immutable. Each Actor, as it completes its processing and sends the message on to the next step in the overall process must include everything that the next Actor will require to complete its task.

Handling External Input

Almost every Real-World Application needs to fetch input from an external source (a source not under its control)

Two choices:

1. Let your Actors block and wait for the fetched information
   - This could easily result in a large number of stalled Actors
   - Incredible resource hog

1. Record the request in a database
   - The database's Supply Clerk Actor is responsible for parsing incoming results and matching them with request records
   - When a result matches up with a request, the Supply Clerk puts a new processing request message in the appropriate queue
   - The Supply Clerk does a periodic scan of outstanding request records. Old records may be an indication of lost requests and may need to be resent
   - Result records without a matching request record are an error and must be flagged to the appropriate Supervisor Actor

Just like no man is an island, very few meaningful business applications are completely self-contained. In most cases, there is a need to access information that's external to our application. If it's a Federal Health Insurance Exchange, we may need to confirm information from an enrollee's Federal Tax return. The IRS is great about this. They collect all your information requests, run a batch job at night and send you back the answers the next day. So it could be as long as 24 hours before your request is satisfied. Now, you could just create a new Actor for each outstanding request. As the answers come back, each Actor could then complete their process and go on to the next job. But if we are enrolling 1 million people per day, that's a lot of stalled Actors standing around waiting. Instead of creating stalled Actors, let's use our Supply Clerk-type Actors. As the IRS Supply Clerk Actor to put the original message into a database. When the IRS results come in, ask the IRS Supply Clerk to retrieve the original message. Add the IRS results to the original message and put it into the queue of the cluster handling the next step in the enrollment process. You also have a nice error detection process. If a message stays in the database more than a day, then the IRS probably lost the original request and we can send it again. By the same token, if the IRS sends us back a result for which we don't have an original message stored, then we have an internal

problem and we immediately raise a red flag.

- Keep Actors small, with one defined action or decision
- Don't build a number of small programs and call them Actors
- Encapsulate common resources in Actors – This eliminates locks and race conditions
- Connect Actors together using Queues
- Monitor workloads by monitoring the size of the Queues – Supervisor Actors create new Actors or message surplus Actors to terminate
- Queues only remove acknowledged Messages
- Use "transaction numbers" on truly critical Messages to make certain actions idempotent
- You can "push" into a Queue, but Actors must ALWAYS "pull" from the Queues

OK, at this point, I think you know enough to go out and architect your own large scale web application. And, I'm asking you to pass this information on to other architects and developers. I never want our government to be embarrassed again like we were with the Heathcare.gov exchange

In Review

- Allow Actors to transparently connect and detach (die) from Queues – no load balancers or registrations
- Keep “state” in the Message between Actors – Each Message is self-sufficient
- Use cryptographic signatures to authenticate Messages and maintain integrity
- Use cryptographic encryption to protect sensitive information in Messages
- Use redundant Actors and Queues for persistence and high availability
- “Map-Reduce” and Consensus (Paxos, Raft, etc.) are your friends
- Use the computing power of the browser – “session-less” Single Page Applications (SPA)
- Use compiled languages – faster, smaller, and more secure

Copies of the slides and the talking points may be downloaded from the Formularity website:

https://formularity.com