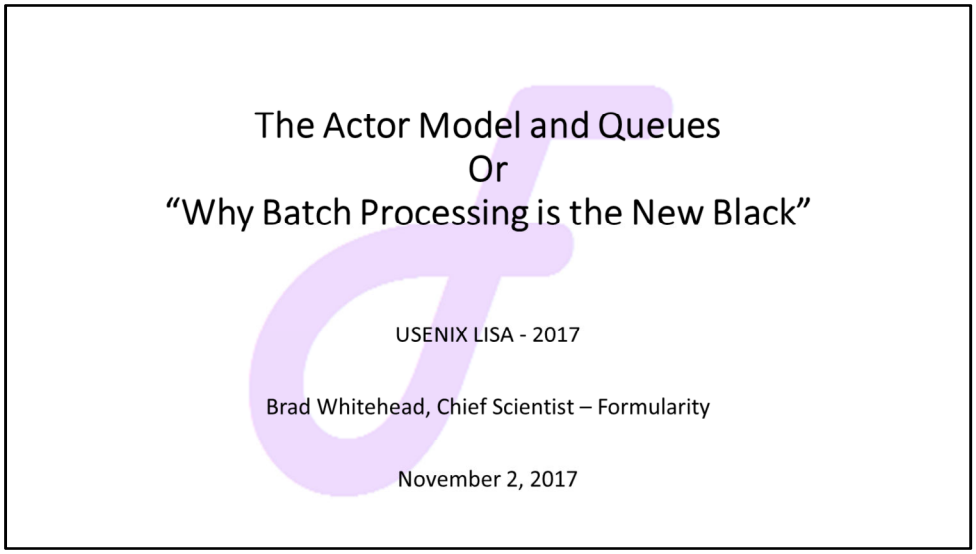


Good afternoon. I'm Brad Whitehead. I'm the Chief Scientist at Formularity. Before I get started, I'd like to thank the folks at Usenix for giving me the opportunity to talk to you today. In particular, I'd like to thank Patrick Cable and Alice Goldfuss, the Invited Talks CoChairs. They have selected an excellent agenda of both interesting and educational presentations. But most importantly, I'd like to thank you for choosing to attend this talk. I hope I make it worth your while! 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 at all times. In addition to our forms being run by our clients in their own 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 enroll millions or tens of 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. I'm not trying to sell your our architecture or our backend software. I will however, sell you electronic forms if you want – come see me at the Reception tonight. I'm here today to share with you what I've learned

in the design and development of infinitely scaling applications. My talk today covers three important elements; The Actor model, Queues, and being polite.

So, at this point, you may be asking - What does Brad know about designing and deploying large applications? Why should I listen to him? Prior to helping to found Formularity, I was a Partner and Master Technology Architect at Accenture. My specialty, if you will, is national level biometric identification and border management systems. I was Accenture's Chief Architect on several large biometric systems, including US VIST (that's the system that where you have to give Customs and Border Protection your fingerprints when you come into the country), the Transportation Worker's Identification Card [TWIC] (you can't work at a US Port or drive a gasoline truck without a TWIC card) , and TSA Pre-Check.

But I'm most proud of the work I did one the Aadhaar program. Anybody here familiar with the Republic of India's Aadhaar program? Aadhaar is one of the greatest personal empowerment programs of all times. It affects and improves the lives of 1.1 billion people in India every day. That's 15% of the world's present population! Every day, my work is helping to improve the lives of over a billion people! It was incredibly fulfilling to work on the Aadhaar program!

Accenture was one of the three original Biometric Service Providers on the Aadhaar program. I took over the role of Chief Architect when we ran into scaling problems. To-date, the Unique Identification Authority of India has enrolled 1.1 billion people in the Aadhaar program; all ten fingers, both irises, and face. Approximately 1 million enrollments are processed per day. The process of enrolling means that we templatize your finger and iris prints and then compare them against all the previous fingerprints and irises we have already enrolled, to make sure you are you and we don't have any fraud in the process. Now, fingerprints and irises can't be indexed or hashed, so you have to do a one-by-one comparison. In SQL terms, you have to do a "full table scan".

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:

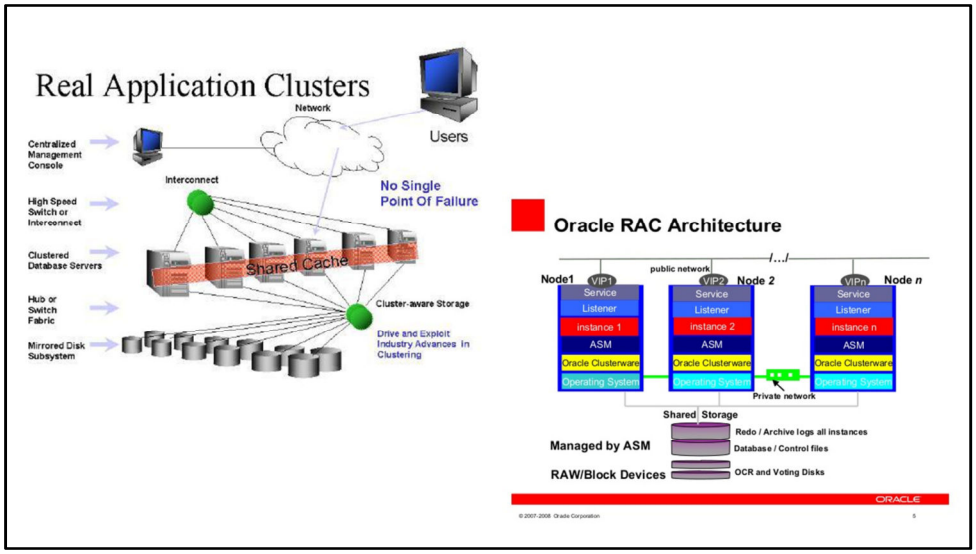
11 Billion Transactions/Second!

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

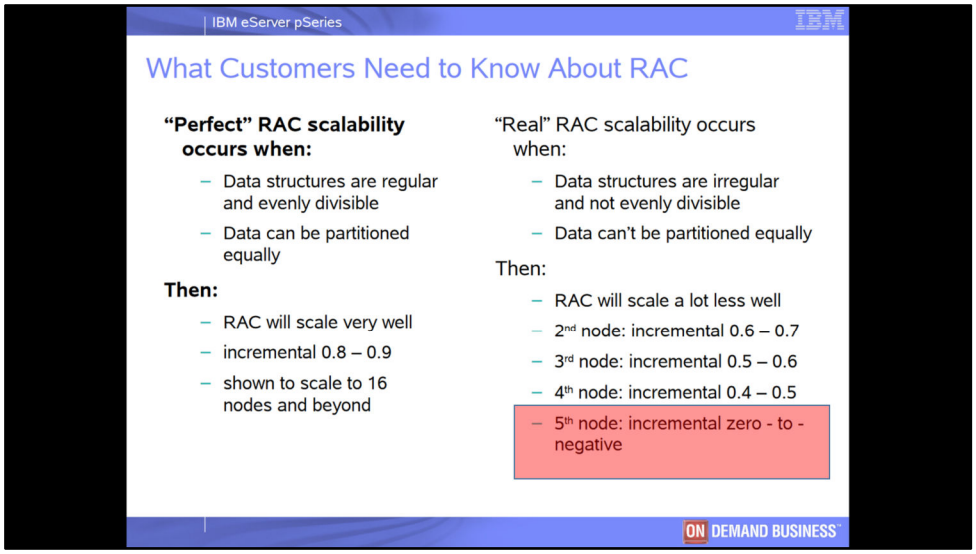
That's a lot of comparisons! Let's look at the "last day" of Aadhaar enrollment. You process the last 1 million enrollees against each of the first 1 billion sets of fingerprints already on file. 1 times 10 to the 6th times 1 times 10 to the 9^{th}. That's 1 times 10 to the 15^{th} or 1 quadrillion biometric template comparisons. That's 11 billion transactions per second!

Initially, we started by 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.



This slide shows a Real Application Cluster, with individual Oracle databases running on separate servers. Basically, you have a number of Oracle databases running and working together. Now, 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. Did anybody attend Baron Schwartz's talk yesterday on the Law of Universal Scalability? He gave an excellent presentation on why things like distributed databases don't scale linearly and why they reach absolute maximums.

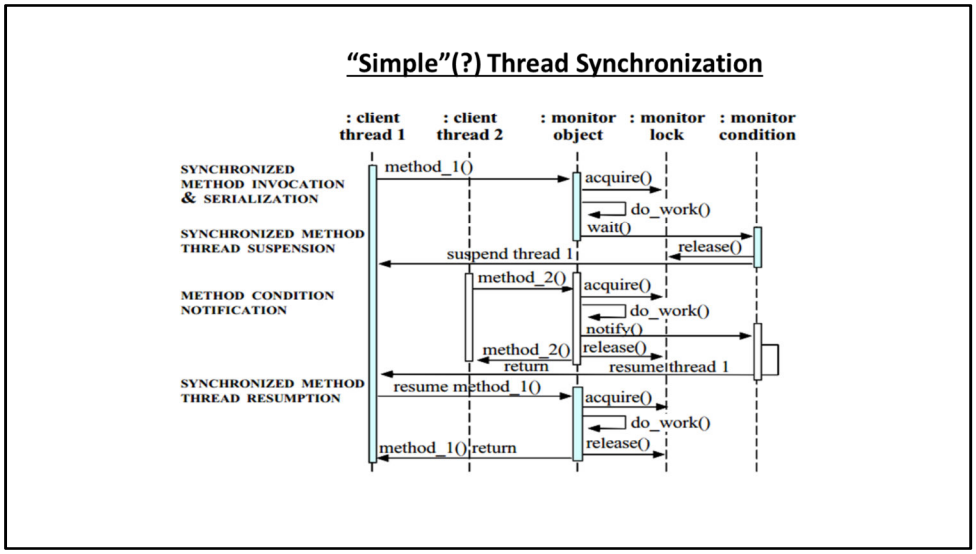
My first job as the Accenture Chief Architect for Aadhaar was to replace the Oracle database-driven workflow system with a new workflow system, based on the Actor Model and queues. This workflow system now easily handles 1 million enrollments per day and we can dial in processing capacity by increasing the number of servers.

Concurrency and Shared Resources

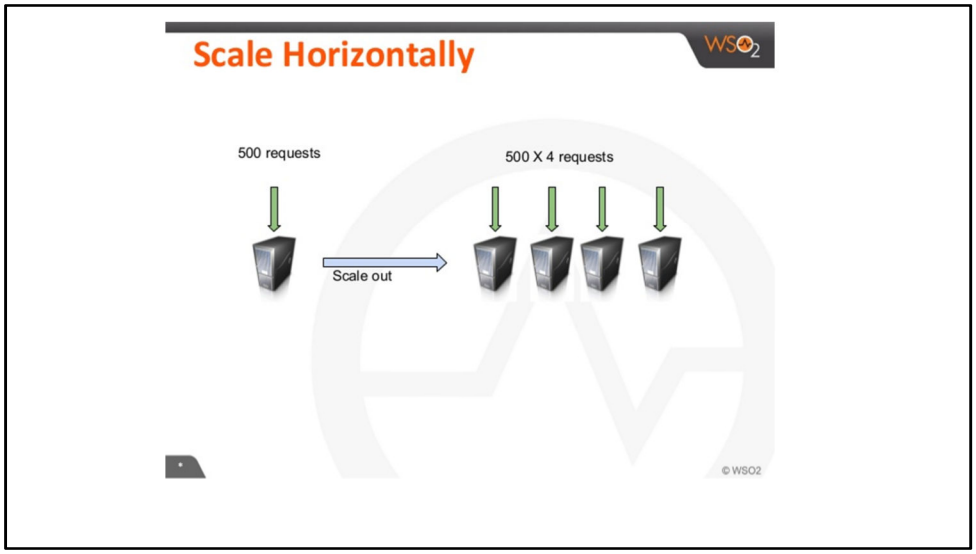
What Can Go Wrong?:
- Corruption
- Race Conditions
- Deadly Embrace
- Blocked Processes and Threads

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

Let's stop and talk about the concept and problems of concurrency for a moment. Computer concurrency is simply the computer doing multiple things at the same time. Used to be, our computers had one processor, one core, and one thread of execution and any illusions of concurrency were just that – illusions. Everything was sequential and life was good. Now days, with multiple cores, we really do have multiple things happening simultaneously. We have true concurrency. We also have multiple threads of execution sharing, or more often than not, competing for resources. So we have developed resource sharing mechanisms like locks and semaphores. Developers of modern, enterprise systems have to worrying about concurrency like race conditions and "deadly embraces." And if they make a mistake, errors and failures occur and since these errors are concurrency-related, they are often hard to replicate and debug..

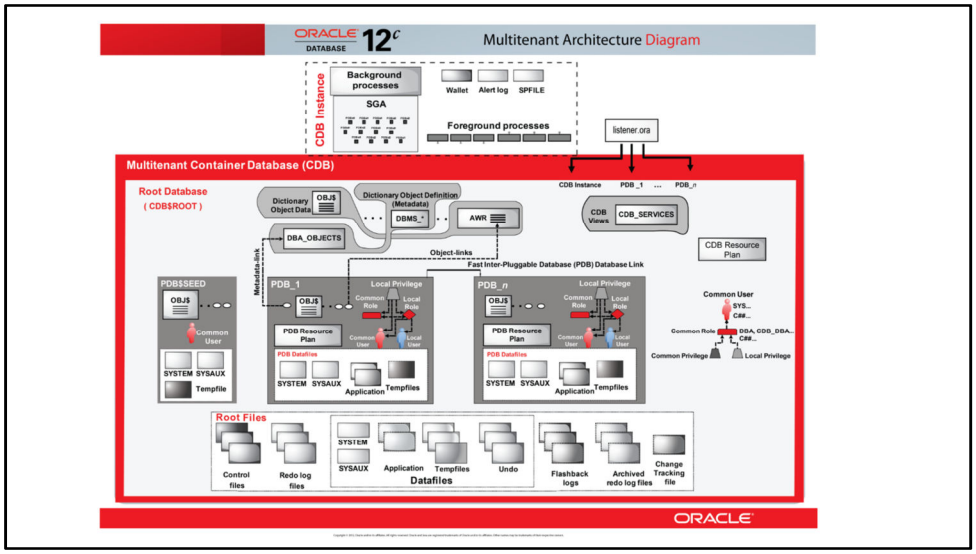


We have developed complex procedures to protect and synchronize shared resources. With complex comes overhead and the potential for errors……



Now, when you reach the core and speed limits of a single computer, how do you scale? Horizontally. We add multiple computers clustered together to increase the parallelism and apparent speed and scale of the application.

Now, 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 and are running complex monolithic applications, we still have all the drawbacks of locks, race conditions, etc.

Carl Hewitt

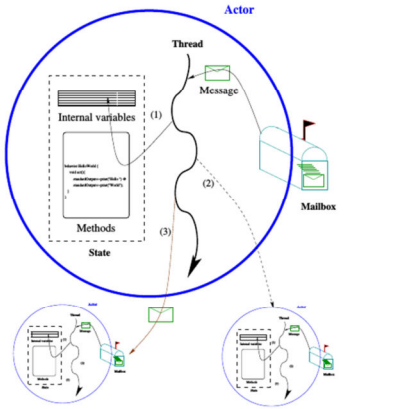


Figure 1.2: In response to a message, an actor can: (1) modify its local state, (2) create new actors, and/or (3) send messages to acquaintances.

The Actor Model

In 1973, Dr. Carl Hewitt at MIT developed the Actor Model. An Actor is an automation that receives external input through messages. Based on a received message, an Actor automation can do one of four things, it can:

Actor Model Methods

1) Read a message;

2) Create new Actors;

3) Send out messages; and

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

Notice, an Actor only communicates through messages. It doesn't directly act on memory, files or other processes. It doesn't compete for or lock resources.

Actor Advantages

1) Actors don't compete for resources

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

3) Actors encapsulate common resources and act as service or resource providers

4) 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

5) Each Actor type can be better tested because of its message passing framework

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

By decomposing our complex monolithic programs into a set of simple services, we can take advantage of the inherent concurrency and isolation aspects of the Actor Model.

So, what are these advantages?

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

8) Multiple Actors can increase availability and/or persistence

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

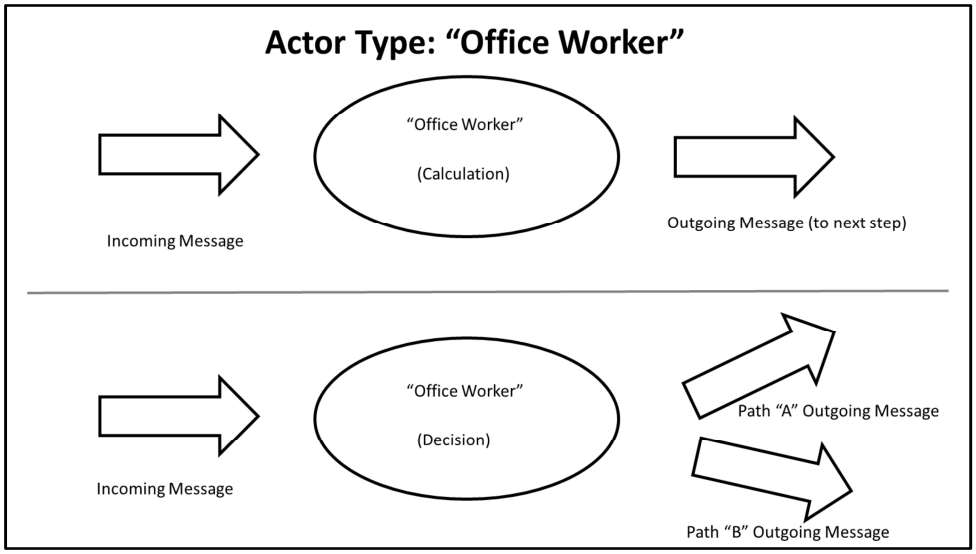
10) If each Actor is run on a separate server, geographical diversity for improved resiliency is possible

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

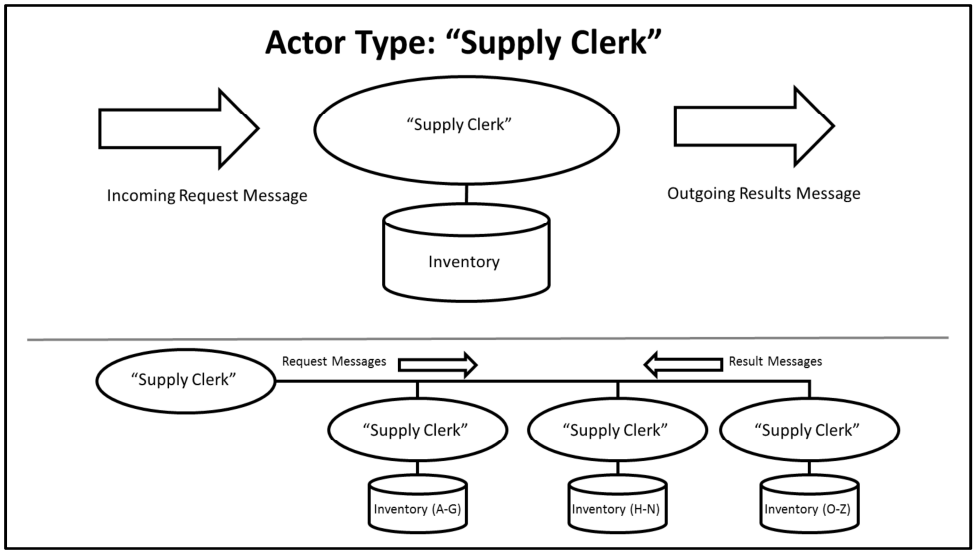
Actors = Microservices

If decomposing complex monolithic programs into smaller, concurrent chunks sounds familiar, well it is – The current term is “microservices”. I think that it’s generally recognized that microservices are a good way to develop reliable, scalable systems, but I’ve seen at lot of confusion about what defines a microservice. The Actor model helps provides the guidance needed to partition functionality into microservices. By developing our microservices as single function, message-driven Actors, we derive all the benefits I just listed.

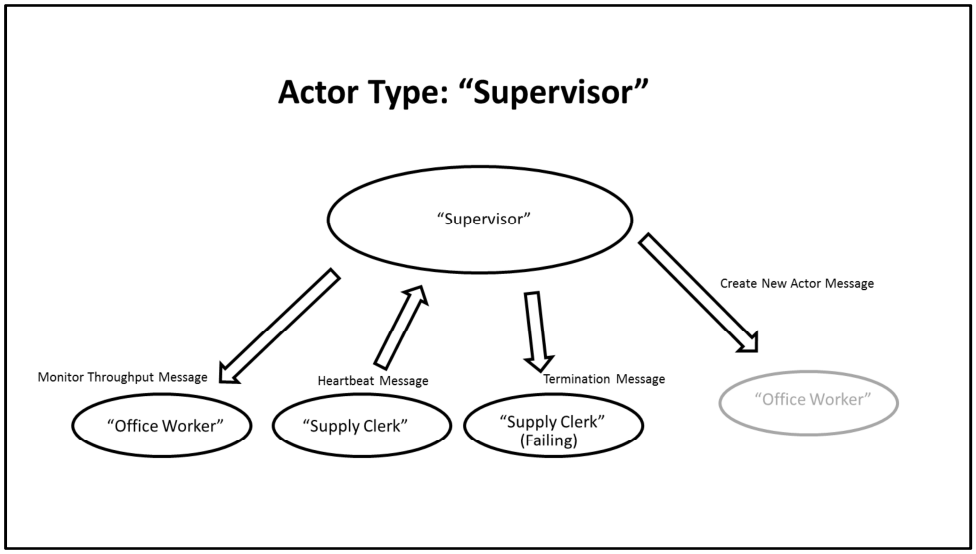
It has been my personal experience over the last 40 years that human nature quite often provides the best way of doing something. So my first step is always to identify how a human would accomplish a given task and then see how it can and should be applied to an automated system. This isn’t foolproof,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



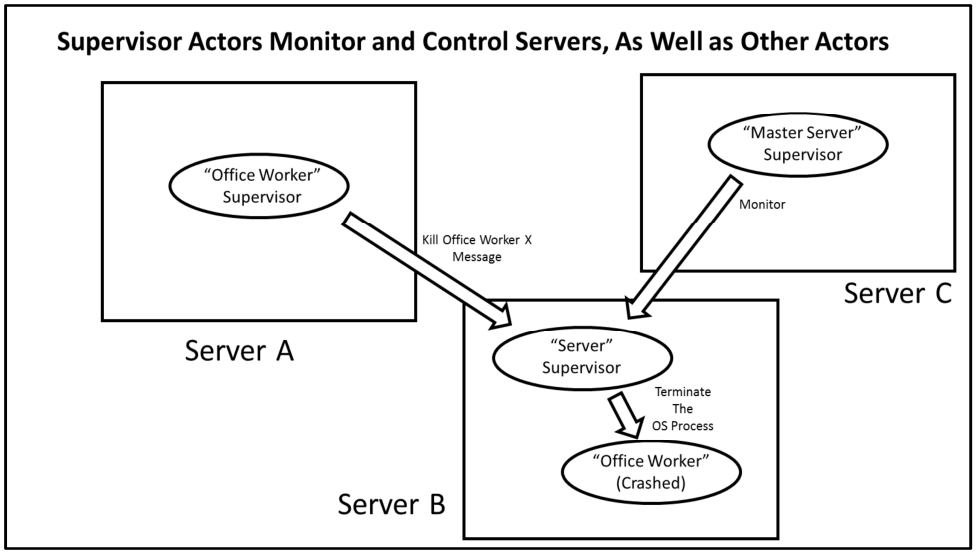
Our first Actor type is the “Office Worker”. Picture a human sitting at a desk, taking papers out of an in-box, reviewing and perhaps stamping them, and then putting them into an out-box. The Office Worker Actor operates just like that. The Office Worker takes messages out of its message queue, performs an action on them and then sends out the results as new messages. The performed process may be linear, such as performing a mathematical calculation; or it may involve a decision. If it’s linear, like the top diagram, 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, shown in the lower diagram, the resulting message may be addressed to different clusters in the application.



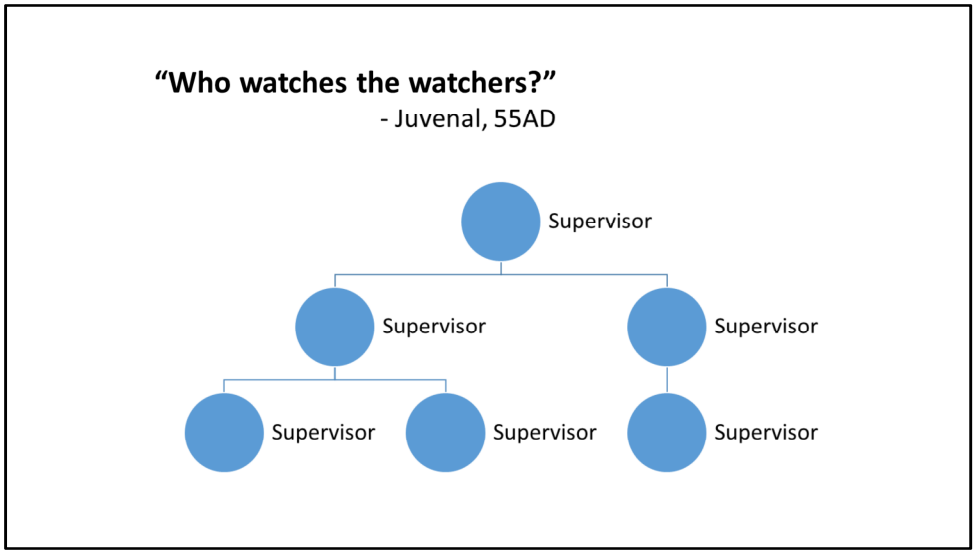
Our next Actor is the Supply Clerk. Remember Radar O’Reilly from the movie and television series M*A*S*H? Radar was responsible for all things administrative. Nobody else could find or do anything administrative but Radar always took care of everything and never (well, hardly ever) made a mistake. The Supply Clerk-type Actor encapsulates common resources. Need to update inventory? Don’t allow multiple Actors to change the inventory. If you do, two or more Actors can 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 replication and sharding 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. 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.



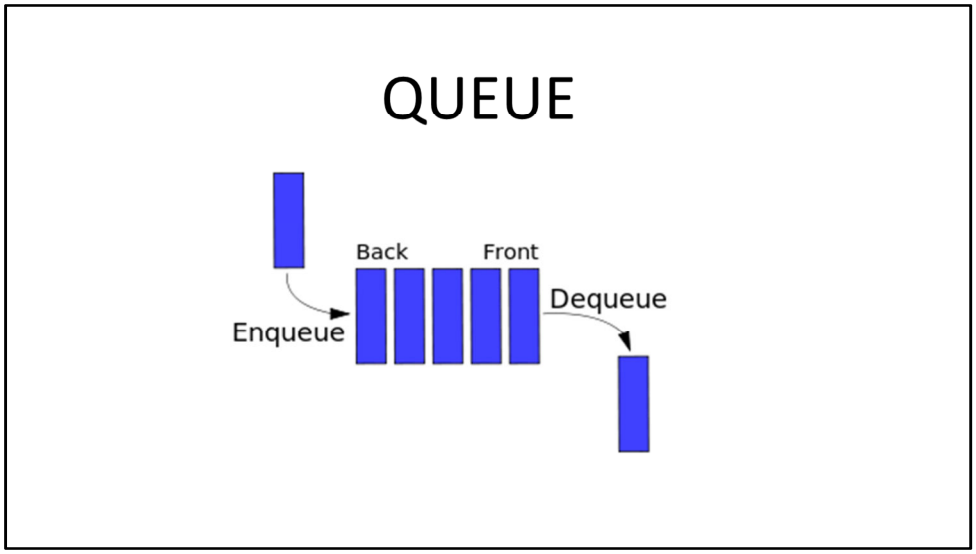
Our third Actor type is the Supervisor. We all know the Supervisor. Doesn't do any useful work but manages the labors of the producers ;-) This doesn't change in the Actor Model. But the Supervisor is responsible for the overall application running smoothly and meeting its processing time requirements. 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 1) 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.



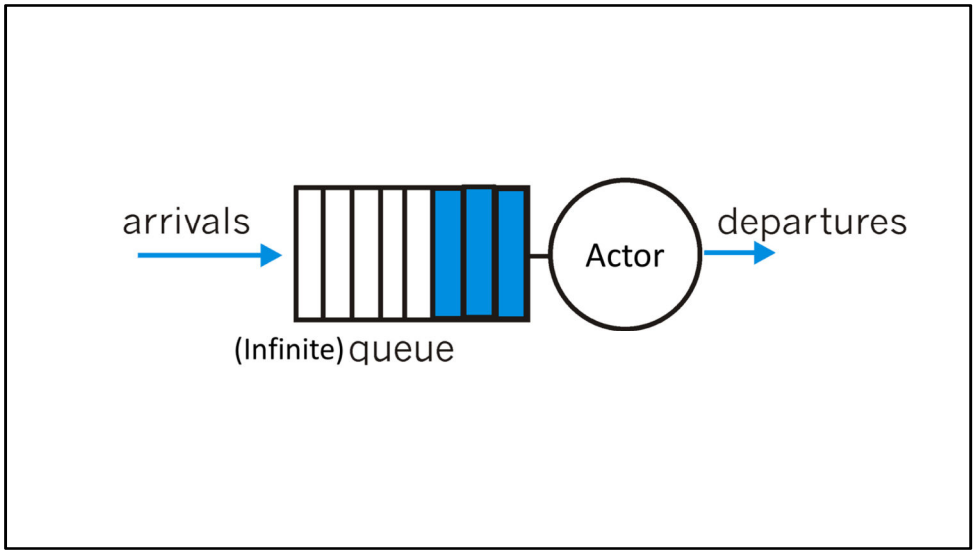
First, I mentioned the Supervisor asking a failing Office Worker-type Actor to terminate itself. This only works if the Office Worker Actor is still interacting. What if it's stalled in a loop and won't read the termination message? The Supervisor Actor has to ask that the operating system process housing the failing Actor be terminated. If the Supervisor Actor is on a different server, 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 each server, and a Master Server Supervisor managing the individual Server Supervisors. Our Office Worker Supervisor asks the Server Supervisor to terminate the failed Office Worker Actor. The Server Supervisor terminates the operating system process that houses the crashed Office Worker. If the Server Supervisor is unresponsive, then our whole server may have crashed or become disconnected from the application. In that case, the Master Server Supervisor is responsible for killing the whole server and restarting a new instance.



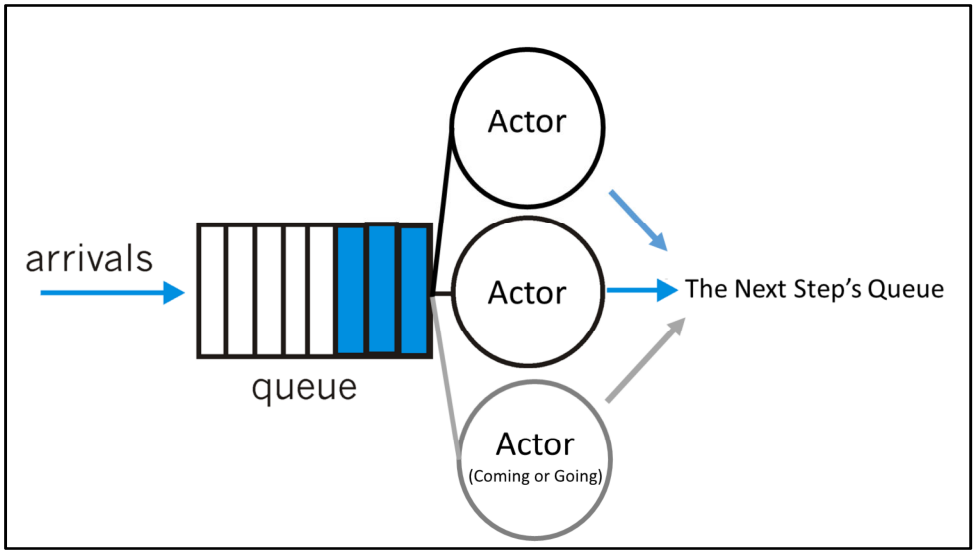
The second example of Supervisor hierarchical cooperation is in supervising the Supervisors. Or to quote the Roman poet Juvenal "Who watches the watchers?" Supervisor Actors have their own Supervisors. At the very top layer of our web service or application, we have a dashboard, showing the state of the servers and the Actors.



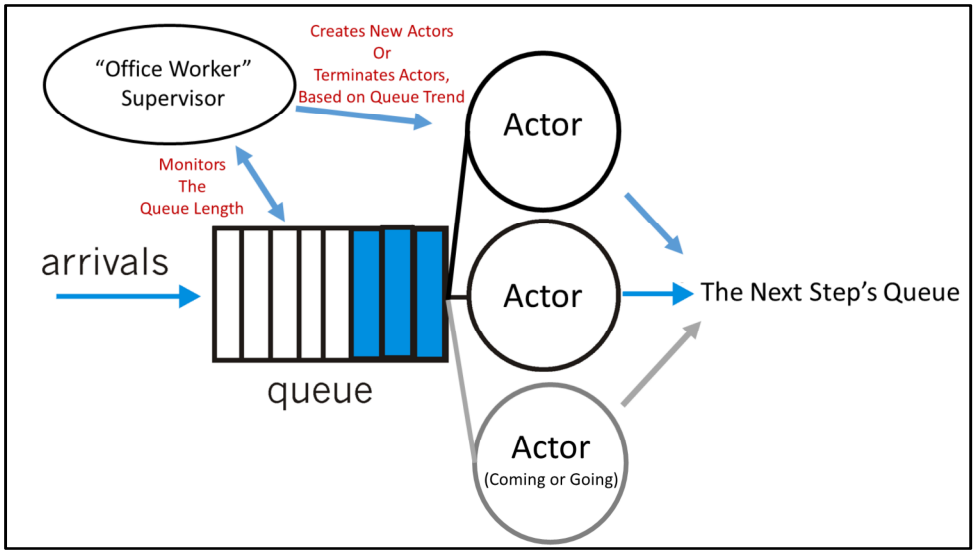
Now, it's important how we connect our Actors.. Part of the key to robust, reliable, and infinitely scalable cloud application is - the Queue. A queue is one of the fundamental data structures we learn in computer science.



What are the advantages of using a queue to connect our Actors, instead of directly sending messages between actors? 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.

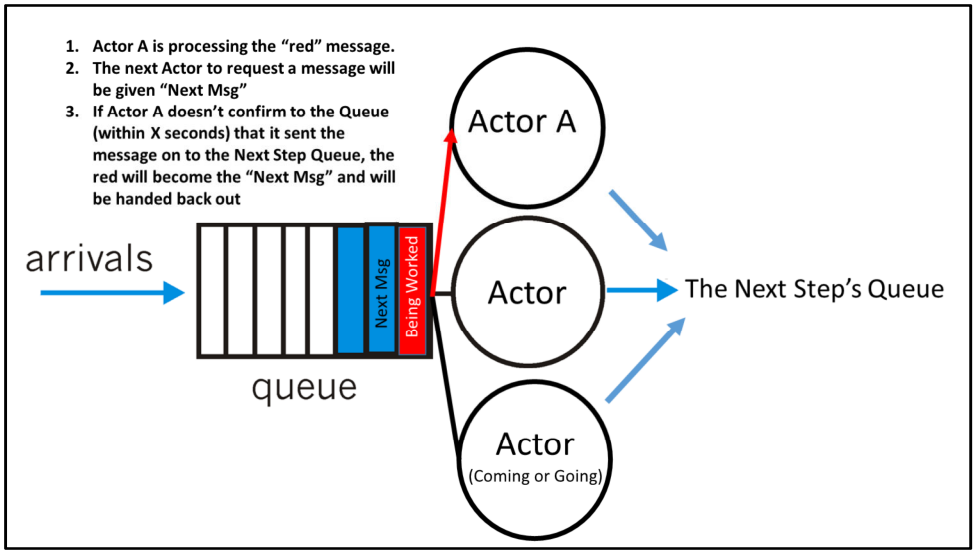


Second, I have talked about clusters of Actors. If one instance of an Actor is insufficient to handle the workload, multiple instances can be created, each picking the next piece of work to be done off the queue. With the queue, instead of individual mailboxes for each Actor, Actors can come and go and they all share in processing the workload. So an Actor doesn'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. Baron Schwartz's presentation yesterday showed us how quickly a server can be overdriven, resulting in degraded performance and failure. The queue gives you a

much better and safer means of monitoring.



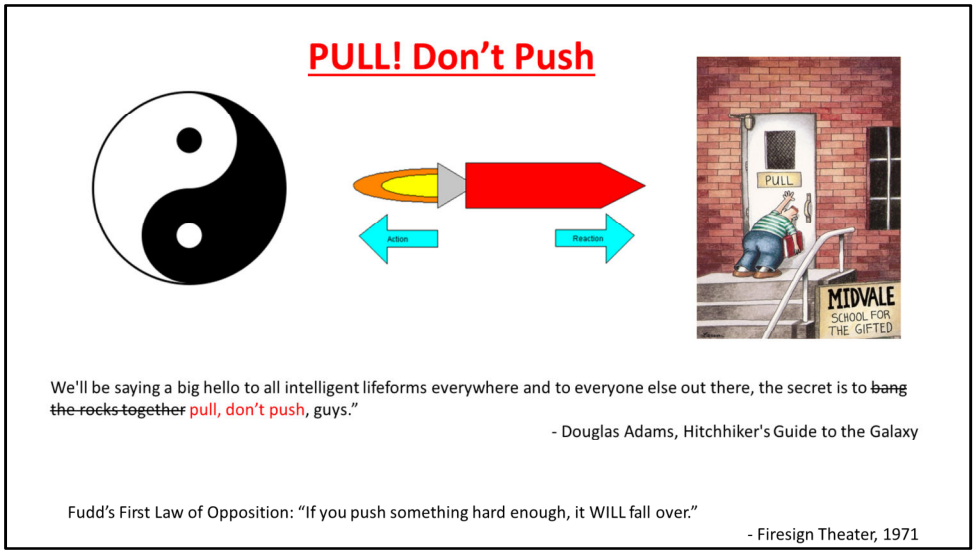
What happens if an Actor dies before completing the processing of a message? Is the message lost? [What do you mean you lost my deposit!!!!?] Or do you devise a scheme where messages have to be retained by their originating Actor until the processing Actor can confirm completion? With this type of complexity, you are setting yourself up for failure. Instead, to paraphrase Obwan Kanobe - "Use the Queue, Luke". When an Actor instance takes a message out of the queue, don't delete 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? Then 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. With this approach, ou 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!!!!]

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.

Cryptography is Cheap! - Use it Everywhere!

- 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

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 these days with the Intel AES-NI instruction set. Use it everywhere!



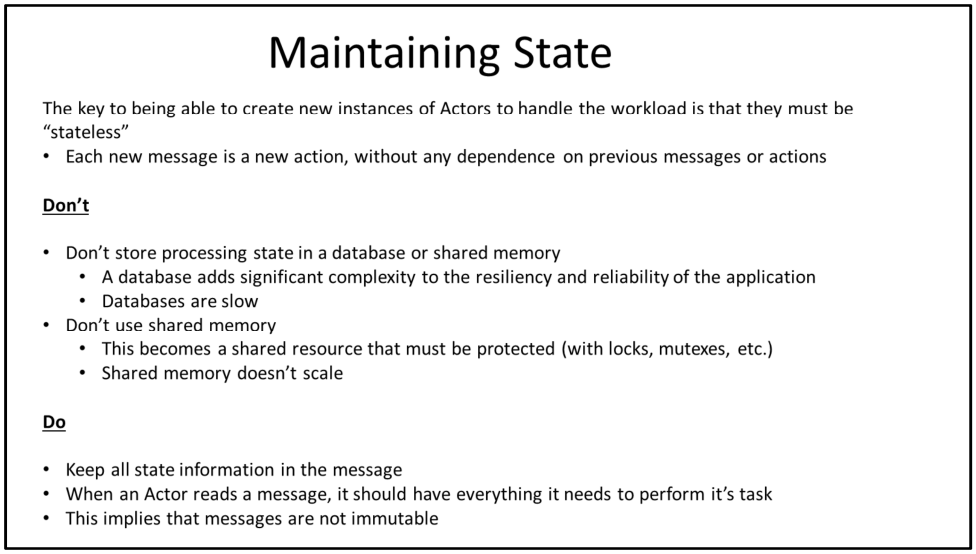
Before I tell you why load balancers are evil, let me talk about the third critical item for a successful large application. Philosophy and Newton tell us that there are always two forces in equal balance. Call them Yin and Yang, or Action and Reaction. Or Push and Pull. In data flow, you can push data or you can pull it. And I'm here to tell you unequivocally that most system architects get it wrong! When you push messages or requests to a server, you can 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 load balancer 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. Just remember, Actors pull from the Queues!

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. John Looney gave an excellent workshop on Tuesday on building distributed systems. In it, he devotes several slides and discussion to the drawbacks of load balancers and balancing algorithms. Let's simplify things. DON'T use load balancers!

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



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 external input.

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. In an Actor model system, **DO** put the state in the message. In other words, when an Actor pulls 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

2. Record the request in a data store (SQL, No-SQL, Key Store, etc.)
   - 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 10 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. Ask the IRS Supply Clerk Actor to put the original message into a data store. 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 data store 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 can immediately raise a red flag.

Key Points

- 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

There's an interesting article in this month's Communications of the ACM. It discusses the journey of HootSuite, a large social media aggregator as they (successfully) transitioned from a classic LAMP stack architecture to one based on Actors and queues, in order to overcome their scaling issues and to increase their reliability and flexibility.

Here's a summary of the key points I'd like you to take away from today's talk…

- 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

OK, at this point, hopefully I've planted the seeds that will help you in architecting your large scale applications. I never want us to be embarrassed again like we were with the Heathcare.gov fiasco! Thank you! Copies of these slides and my talking notes will be available on the Formularity website later today, as well as through the LISA 2017 Open Access website. Are there any questions?...