Fault Tolerance in Distributed Databases/Systems

Rishabh sahu [1] Pamu Nagarjuna [2] Bala Vineeth[3] Rajshekhar Reddy[4]

***Abstract*— Enterprise application downtimes, server crashes because of huge load are very scary to hear as they inccur huge loses to the corresponding organization. Most of the big enterprice applications are concurrent and parallel as they have to serve large number of users offering mission critical services. In this scenario fault tolerance and thinking what can go wrong while building the application can help us a lot. Our strategy for building mission critical applications and dealing with failure has to change. So, to solve this problem of coping up with failure and allowing graceful degradation of the services, we propose a set of popular patterns which can not only avoid failure, but can deal with it by embracing the failure. Self healing is one other attribute that can help us. Strategies like fail fast and circuit breakers can help stop the pressure on the degrading components.**

I. INTRODUCTION

Fault tolerance is very critical for enterprise applications and its a prime concern. The way we build applications has been changing. Large Enterprise applications are both concurrent, parallel and also are distributed to serve large number of users across the globe. In this setting faults are common and bound to happen. Fault may manifest in various ways like network failure, undeteministic delay of messages, hardware failures, natural calamaties, hackers attack and other failures due to human interventions. So, instead of fearing failure application developers should understand that there is no such thing in real world and try to embrace failure and make it part of the application codebase. Unfortunately, there are no tools, right level of abstractions to make the failure handling code interesting and enjoying. So, developers consider it boring and do something called as defensive programming and be confident about failure handling, but that may not help you with that. What we want to build is something that we can reason about and be confident that, this is how it behaves in case of failure and deterministicly handle failure cases. These are some of the techniques, suggestions and patterns which can help us deal with failure in a distributed setting. The traditional way of dealing with distributed systems was using RPC style of programming. RPC means remote procedural call. Using RPC we are trying to abstract over network latency, network failure and message marshalling. So, programmer has to make a method call just like a local method call and all responsibility of network call, message marshalling is taken care my the proxy object behind the scenes offering great flexibility to the user.RPC considers a remote call like a local call and hence ignores partial failure, latency.

II. TRADITIONAL RPC

A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called remote invocation or remote method invocation.
The ”traditional” RPC services are not well suited for the internet (e.g. dynamically choosing port numbers causing problems with firewalls). New RPC-like protocols are called Web services and often use HTTP for transport and XML for the content (i.e., using XML as the standardized format for marshalling arguments and results).

- What if request is lost
- What if response is lost
- Caller is held hostage by the callee

III. CIRCUIT BREAKER

Circuit breaker is a design pattern in modern software development.circuit breaker is used to detect failures and encapsulates logic of preventing a failure to reoccur constantly (during maintenance, temporary external system failure or unexpected system difficulties). Assume that your application connects to a database 100 times per second and the database fails. You do not want to have the same error reoccur constantly. You also want to handle the error quickly and gracefully without waiting for TCP connection timeout.Generally Circuit Breaker can be used to check the availability of an external service. An external service can be a database server or a web service used by the application.circuit breaker detects failures and prevents the application from trying to perform the action that is doomed to fail (until its safe to retry).

A. Why are they used?

A circuit breaker is used to provide stability and prevent cascading failures in distributed systems. These should be used in conjunction with judicious timeouts at the interfaces between remote systems to prevent the failure of a single component from bringing down all components.

As an example, we have a web application interacting with a remote third party web service. Let’s say the third party has oversold their capacity and their database melts down under load. Assume that the database fails in such a way that it takes a very long time to hand back an error to the third party web service. This in turn makes calls fail after a long period of time. Back to our web application, the users have noticed that their form submissions take much longer

seeming to hang. Well the users do what they know to do which is use the refresh button, adding more requests to their already running requests. This eventually causes the failure of the web application due to resource exhaustion. This will affect all users, even those who are not using functionality dependent on this third party web service.

Introducing circuit breakers on the web service call would cause the requests to begin to fail-fast, letting the user know that something is wrong and that they need not refresh their request. This also confines the failure behavior to only those users that are using functionality dependent on the third party, other users are no longer affected as there is no resource exhaustion. Circuit breakers can also allow savvy developers to mark portions of the site that use the functionality unavailable, or perhaps show some cached content as appropriate while the breaker is open.

IV. Throttling

Suppose you are writing an application that makes HTTP requests to an external web service and that this web service has a restriction in place: you may not make more than 10 requests in 1 minute. You will get blocked or need to pay if you dont stay under this limit. In such a scenario you will want to employ a message throttler.

This extension module provides a simple implementation of a throttling actor, the TimerBasedThrottler.

A. The guarantees

TimerBasedThrottler uses a timer internally. When the throttlers rate is 3 msg/s, for example, the throttler will start a timer that triggers every second and each time will give the throttler exactly three ”vouchers”; each voucher gives the throttler a right to deliver a message. In this way, at most 3 messages will be sent out by the throttler in each interval.
It should be noted that such timer-based throttlers provide relatively weak guarantees:

- Only start times are taken into account. This may be a problem if, for example, the throttler is used to throttle requests to an external web service. If a web request takes very long on the server then the rate observed on the server may be higher.
- A timer-based throttler only makes guarantees for the intervals of its own timer. In our example, no more than 3 messages are delivered within such intervals. Other intervals on the timeline, however, may contain more calls.

V. Event Sourcing

What is event sourcing?

Storing all the changes (events) to the system, rather than just its current state.

Why haven’t I heard of event stores before?

You have. Almost all transactional RDBMS systems use a transactional log for storing all changes applied to the database. In a pinch, the current state of the database can be recreated from this transaction log. This is a kind of event store. Event sourcing just means following this idea to its conclusion and using such a log as the primary source of data.

A. Advantages

- Ability to put the system in any prior state. Useful for debugging. (I.e. what did the system look like last week?)
- Having a true history of the system. Gives further benefits such as audit and traceability. In some fields this is required by law.
- We mitigate the negative effects of not being able to predict future needs, by storing all events and being able to create arbitrary read-side projections as needed. This allows for more nimble responses to new requirements.
- The kind of operations made on an event store is very limited, making the persistence very predictable and thus easing testing.
- Event stores are conceptually simpler than full RDBMS solutions, and it’s easy to scale up from an in-memory list of events to a full-featured event store.

B. Snapshotting

An optimization where a snapshot of the aggregate’s state is also saved (conceptually) in the event queue every so often, so that event application can start from the snapshot instead of from scratch. This can speed things up. Snapshots can always be discarded or re-created as needed, since they represent computed information from the event stream.

Typically, a background process, separate from the regular task of persisting events, takes care of creating snapshots.

Snapshotting has a number of drawbacks related to re-introducing current state in the database. Rather than assume you will need it, start without snapshotting, and add it only after profiling shows you that it will help.

C. Handling large event

Events are usually quite small, and you can easily store, index, and search millions of them on a low-end relational database.

That said, it’s always good to plan ahead, and pick a serialization format that serves you well in terms of size. JSON tends to be smaller than the corresponding XML, for example.

If you feel the need to algorithmically compress your events, that’s also an option. Google’s protocol-buffers are a modern example of a compressed serialization to use.

For the cases where you actually literally run out of hard drive space: disks are cheap nowadays. Consider saving historical events in some permanent storage. The events carry important business value; do not throw them away.

If the event store outgrows a single machine, then it is easy to shard first by aggregate type, and with a little content-based routing even at the level of aggregates themselves.

VI. Bulk heads

In orderto manage failure we need a way to isolate it so it doesnt spread to other healthy components, and to observe it

so it can be managed from a safe point outside of the failed context. One pattern that comes to mind is the bulkhead pattern,illustrated by the picture,in which a system is built up from safe compartments so that if one of them fails the other ones are not affected.This prevents the classic problem of cascading failures and allows the management of problems in isolation. The event-driven model,which enables scalability, also has the necessary primitives to realize this model of failure management.The loose coupling in an event-driven model provides fully isolated components in which failures can be captured together with their context, encapsulated as messages, and sent off to other components that can inspect the error and decide how to respond. This approach creates a system where business logic remains clean, separated from the handling of the unexpected,where failure is modeled explicitly in orderto be compartmentalized, observed, managed and configured in a declarative way, and where the system can heal itself and recover automatically.It works best if the compartments are structured in a hierarchical fashion, much like a large corporation where a problem is escalated upwards until a level is reached which has the powerto deal with it. The beauty of this model is that it is purely event-driven, based upon reactive components and asynchronous events and therefore location transparent.In practice this means that it works in a distributed environment with the same semantics as in a local context.

VII. Let it Crash Model

- Embrace failure as a natural state in the life-cycle of the application.
- Instead of trying to prevent it,manage it.
- Process supervision.
- Supervisor hierarchies (from Erlang)

Designing fault tolerant systems is extremely difficult. You can try to anticipate and reason about all of the things that can go wrong with your software and code defensively for these situations, but in a complex system it is very likely that some combination of events or inputs will eventually conspire against you to cause a failure or bug in the system.

In certain areas of the software community such as Erlang and Akka, theres a philosophy that rather than trying to handle and recover from all possible exceptional and failure states, you should instead simply fail early and let your processes crash, but then recycle them back into the pool to serve the next request. This gives the system a kind of self healing property where it recovers from failure without ceremony, whilst freeing up the developer from overly defensive error handling.

I believe that implementing let it crash semantics and working within this mindset will improve almost any application not just real time Telecoms system where Erlang was born. By adopting let it crash, redundancy and defence against errors will be baked into the architecture rather than trying to defensively anticipate scenarios right down in the guts of the code. It will also encourage you to implement more redundancy throughout your system.

Also ask yourself, if the components or services in your application did crash, how well would your system recover with or without human intervention? Very few applications will have a full automatic recoverability property, and yet implementing this feels like relatively low hanging fruit compared to writing 100 percentage fault tolerant code.

So how do we start to put this into practice?

At the hardware level, you can obviously look towards the Google model of commodity servers, whereby the failure of any given server supporting the system does not lead to a fatal degradation of service. This is easier in the cloud world where the economics encourage us to use a larger number of small virtualised servers. Just let them crash and design for the fact that servers can die at a moments notice.

Your application might be comprised of different logical services. Think a user authentication service or a shopping cart system. Design the system to let entire services crash. Where appropriate, your application should be able to proceed and degrade gracefully whilst the service is not available, or to fall back onto another instance of the service whilst the first one is recycling. Nothing should be in the critical code path because it might crash!

Ideally, your distributed system will be organised to scale horizontally across different server nodes. The system should load balance or intelligently route between processes in the pool, and different nodes should be able to join or leave the pool without too much ceremony or impact to the application. When you have this style of horizontal scalability,let nodes within your application crash and rejoin the pool when theyre ready.

What if we go further and implement let it crash semantics for our infrastructure?

For instance, say we have some messaging system or message broker that transports messages between the components of your application. What if we let that crash and come back online later. Could you design the application so that this is not as fatal as it sounds, perhaps by allowing application components to write to or dynamically switch between two message brokers?

Distributed NoSQL data stores gives us let it crash capability at the data persistence level. Data will be stored in some distributed grid of nodes and replicated to at least 2 different hardware nodes. At this point, its easier to let database nodes crash than try to achieve 100 percentage uptime.

At the network level, we can design topologies such that we do not care if routers or network links crash because theres always some alternate route through the network. Let them crash and when they come back the optimal routes will be there ready for our application to make use of again in future.

Let it crash is more than simple redundancy. Its about implementing self recoverability of the application. Its about putting your site reliability efforts into your architecture rather than low level defensive coding. Its about decoupling your application and introducing asynchronicity in recognition that things go wrong in surprising ways. Ironically, sitting back and cooly letting your software crash can lead

to better software!

VIII. CONCLUSIONS

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100 percentage uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterdays software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback. Reactive Systems are: Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction. Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures. Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms. Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead. Large systems are composed of smaller ones and therefore depend on the Reactive properties of their constituents. This means that Reactive Systems apply design principles so these properties apply at all levels of scale, making them composable. The largest systems in the world rely upon architectures based on these properties and serve the needs of billions of people daily. It is time to apply these design principles consciously from the start instead of rediscovering them each time.

References are important to the reader; therefore, each citation must be complete and correct. If at all possible, references should be commonly available publications.

REFERENCES

[1] http://cqrs.nu/Faq/event-sourcing
[2] http://doc.akka.io/docs/akka/snapshot/common/circuitbreaker.html
[3] http://wiki.wireshark.org/RPC
[4] http://en.wikipedia.org/wiki/Remoteprocedurecall
[5] http://www.reactivemanifesto.org/pdf/the-reactive-manifesto-1.1.pdf
[6] http://www.reactivemanifesto.org/
[7] http://www.slideshare.net/jboner/scalability-availability-stability-patterns
[8] http://web.archive.org/web/20090430014122/http://nplus1.org/articles/a-crash-course-in-failure/
[9] http://architects.dzone.com/articles/increasing-system-robustness
[10] http://doc.akka.io/docs/akka/2.3.7/java/lambda-fault-tolerance.html
[11] http://doc.akka.io/docs/akka/2.3.7/java/persistence.html
[12] http://doc.akka.io/docs/akka/2.3.7/common/cluster.html
[13] http://steve.vinoski.net/pdf/IEEE-ConvenienceOverCorrectness.pdf
[14] http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
[15] http://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf
[16] http://www.bailis.org/papers/hat-hotos2013.pdf
[17] http://macs.citadel.edu/rudolphg/csci604/ImpossibilityofConsensus.pdf
[18] http://the-paper-trail.org/blog/a-brief-tour-of-flp-impossibility/
[19] http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf
[20] http://cs.brown.edu/ mph/HerlihyW90/p463-herlihy.pdf