ADESINA ALAMEEN.B.

18/SCI01/099

CSC 407

NET CENTRIC COMPUTING

FALACIES OF DISTRIBUTED SYSTEMS

The 8 fallacies are:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Let's go through each fallacy, discussing the problem and potential solutions.

**1. The Network Is Reliable**

**The Problem**

**Calls over a network will fail.**

Most of the systems today make calls to other systems. Are you integrating with 3rd party systems (payment gateways, accounting systems, CRMs)? Are you doing web service calls? What happens if a call fails? If you're querying data, a simple retry will do. But what happens if you're sending a command?

What happens if we receive an HTTP timeout exception? If the server did not process the request, then we can retry. But, if it did process the request, we need to make sure we are not double charging the customer. You can do this by making the server idempotent. This means that if you call it 10 times with the same charge request, the customer will be charged only once. If you're not properly handling these errors, you're system is nondeterministic. Handling all these cases can get quite complex really fast.

### Solutions

So, if calls over a network can fail, what can we do? Well, we could **automatically retry**. Queuing systems are very good at this. They usually use a pattern called store and forward. They store a message locally, before forwarding it to the recipient. If the recipient is offline, the queuing system will retry sending the message. MSMQ is an example of such a queuing system.

But this change will have a big impact on the design of your system. You are moving from a request/response model to fire and forget. Since you are not waiting for a response anymore, you need to change the user journeys through your system. You cannot just replace each web service call with a queue send.

### Conclusion

You might say that networks are more reliable these days - and they are. But stuff happens. Hardware and software can fail - power supplies, routers, failed updates or patches, weak wireless signals, network congestion, rodents or sharks. Yes, sharks: Google is reinforcing undersea data cables with Kevlar after a series of shark bites.

And there's also the people side. People can start DDOS attacks or they can sabotage physical equipment.

Does this mean that you need to drop your current technology stack and use a messaging system? Probably not! You need to weigh the risk of failure with the investment that you need to make. You can minimize the chance of failure by investing in infrastructure and software. In many cases, failure is an option. But you do need to consider failure when designing distributed systems.

## 2. Latency Is Zero

### The Problem

**Calls over a network are not instant.**

There is a seven-orders-of-magnitude difference between in-memory calls and calls over the internet. Your application should be network aware. This means you should clearly separate local calls from remote calls.

Without further inspection, this looks fine. But, there are two remote calls. Line 2 makes one call to get a list of document summaries. At line 5 there is another call that retrieves more information about each document. This is a classic Select n+1 problem. In order to account for the network latency, you should return all the required data in one call. The general advice is that local calls can be fine-grained, but remote calls should be more coarse-grained. This is why the idea of distributed objects and "network transparency" died. But, even though everybody agrees that distributed objects are a bad idea, some people still think that lazy loading is always a good idea:

### Solutions

#### Bring Back All the Data You Might Need

If you do a remote call, ensure that you **bring back all the data you might need**. Network communication should not be chatty.

#### Move the Data Closer to the Clients

Another possible solution is to **move the data closer to the clients**. If you're using the cloud, choose availability zone carefully, depending on your client's location. Caching can also help minimize the number of network calls. For static content, Content Delivery Networks (CDNs) are another good option.

#### Invert the Flow of Data

Another option for removing the remote calls is to **invert the flow of data**. Instead of querying other services, we can use Pub/Sub and store the data locally. This way, we'll have the data when we need it. Of course, this introduces some complexity, but it can be a good tool in the toolbox.

### Conclusion

Although latency might not be an issue in LANs, when you move to WANs or the Internet, you'll notice the delay. This is why it's important for network calls to be clearly separated from in-memory calls. You should keep this in mind when adopting the microservices architectural pattern. You shouldn't just replace local calls with remote calls. Chances are this will turn your system into a distributed big ball of mud.

## 3. Bandwidth Is Infinite

### The Problem

**Bandwidth is limited.**

Bandwidth is the capacity of a network to send data over a period of time. Up until now, I've not found it to be a problem, but I can see why it could be an issue in certain conditions. Although bandwidth has improved over time, the amount of data that we send has increased too. Video streaming or VoIP will need more bandwidth than apps that are passing simple DTOs over the network. Bandwidth is even more important for mobile apps, so developers need to think about it when designing their backend APIs.

Using an ORM incorrectly can hurt too. I've seen examples of developers calling .ToList() too early in a query, thus loading an entire table in memory.

### Solutions

#### Domain-Driven Design Patterns

So how can we make sure we don't bring too much data? Domain-Driven Design patterns can help:

* First, you should not strive for a single, enterprise-wide domain model. You should **partition your domain into bounded contexts.**
* To avoid large and complex object graphs inside a bounded context, you could use the Aggregate pattern. Aggregates ensure consistency and define transactional boundaries.

#### Command and Query Responsibility Segregation

We sometimes load complex object graphs because we need to display parts of it on a screen. If we do this in lots of places, we end up with a large and complicated model that is suboptimal for both writing and reading. Another approach could be to**use Command and Query Responsibility Segregation** - CQRS. This means splitting the domain model in two:

* The **write model** will ensure invariant hold true and the data is consistent. Since the write model doesn't care about view concerns, it can be kept small and focused.
* The **read model** is optimized for view concerns, so we can retrieve all the data that is required for a specific view (e.g. a screen in our app).

### Conclusion

There is a tension between the second fallacy, **latency is not 0**, and the third fallacy,**bandwidth is infinite**. You should transfer more data to minimize the number of network round trips. You should transfer less data to minimize bandwidth usage. You need to balance these two forces and find the right amount of data to send over the wire.

Although you might not hit the bandwidth limitation often, thinking about the data that you transfer is important. Less data is easier to understand. Less data means less coupling. So transfer only the data that you might need.

## 4. The Network Is Secure

### The Problem

**The network is insecure.**

This is one assumption that gets more media coverage than the others. Your system is only as secure as your weakest link. The bad news is that there are a lot of links in a distributed system. You're using HTTPS, except when communicating with that 3rd party legacy system that doesn't support it. You're reviewing your own code, looking for security issues, but are using open source libraries that might be a risk. An OpenSSL vulnerability allowed people to steal data protected by SSL/TLS. A bug in Apache Struts allowed attackers to execute code on the server. Even if you're protecting against all of that, there still is the human factor. A malicious DBA can "misplace" a database backup. The attackers of today have a lot of computing power in their hands and a lot of patience. So the question is not **if** they're going to attack your system, but **when**.

### Solutions

#### Defense in Depth

You should use a layered approach to secure your system. You need different security checks at the network, infrastructure and application level.

#### Security Mindset

Keep security in mind when designing your system. The top ten vulnerabilities list has not changed that much in the last 5 years. You should follow best practices for secure software design and review code for common security flaws. You should regularly search 3rd party libraries for new vulnerabilities. The list of Common Vulnerabilities and Exposures can help.

#### Threat Modeling

Threat modeling is a systematic approach of identifying possible security threats in a system. You first identify all the assets in your system (user data in the database, files, etc.) and how they are accessed. After that, you identify possible attacks and start executing them. I recommend reading Chapter 2 in Advanced API Security for a good overview of Threat Modelling.

### Conclusion

The only secure system is one that is powered off, not connected to any network (and ideally cast in a block of concrete). What a useful system that it! The truth is that security is hard and expensive. There are a lot of components and links in a distributed system and each one of them is a possible target for malicious users. The business needs to balance the risk and probability of an attack with the cost of implementing prevention mechanisms.

Attackers have a lot of patience and computing power at their hands. We can prevent certain types of attacks by using threat modeling, but we can't guarantee 100% security. So it's a good idea to make this clear to the business, decide together how much to invest in security and have a plan for when a security breach does happen.

## 5. Topology Doesn't Change

### The Problem

**Network topologies change constantly.**

Network topology changes all the time. Sometimes it changes for accidental reasons - when your app server goes down and you need to replace it. Many times, it's deliberate — adding new processes on a new server. Nowadays, with cloud and containers on a rise, this is even more visible. Elastic scaling - the ability to add or remove servers depending on the workload - requires a certain degree of network agility.

### Solutions

#### Abstract the Physical Structure of the Network

The first thing you need to do is to abstract the physical structure of the network. There are several ways in which you can do that:

* Stop hardcoding IPs - You should prefer using hostnames. By using URIs we are relying on the DNS to resolve the hostname to an IP.
* When DNS is not enough (e.g. when you need to map an IP and a port), then use discovery services.
* Service Bus frameworks can also provide location transparency.

#### Cattle, Not Pets

By treating your servers as cattle, not pets you are ensuring that no server is irreplaceable. This bit of wisdom helps you get into the right mindset: any server can fail (thus changing the topology), so you should automate as much as you can.

#### Test

One last piece of advice is to test your assumptions. Stop a service or shut down a server and see if your system is still up and running. Tools like Netflix's Chaos Monkey take this up a notch, by randomly shutting down VMs or containers in your production environment. By bringing the pain forward, you are more motivated to build a more resilient system that can handle topology changes.

### Conclusion

Ten years ago, most topologies didn't change that often. But when it did, it probably happened in production and introduced some downtime. Nowadays, with cloud and containers on the rise, it's hard to ignore this fallacy. You need to be prepared for failure and test for it. Don't wait for it to happen in production!

## 6. There Is One Administrator

### The Problem

**There is no one person who knows everything.**

Well, this one seems obvious. Of course, there is no one person who knows everything. Is this a problem? As long as the application runs smoothly, it isn't. But, when something goes wrong, you'll need to fix it. Because so many people touched the app, the one who knows how to fix the problem might not be there.

There are many things that could go wrong. One example is configuration. Today's applications store configurations in multiple stores: config files, environment variables, database, command line arguments. There is no one person that knows what is the effect of every possible config value.

Another thing that could go wrong is system upgrades. A distributed application has many moving parts and you need to make sure they are synchronized. For example, you need to make sure that the current version of the code works with the current version of the database. Nowadays there is a focus on DevOps and Continuous Delivery. But supporting zero downtime deployment is no easy feat.

But, at least these things are under your control. Many apps interact with 3rd party systems. This means that, if they go down, there isn't much that you can do. So, even if you had one administrator for your system, you still can't control 3rd party systems.

### Solutions

#### Everyone Should Be Responsible for the Release Process

This means involving Ops people or system administrators from the start. Ideally, they would be part of the team. Keeping the system administrator informed of your progress early on can help you spot constraints. For example, the production environment might have different configurations, security restrictions, firewall rules or available ports than a development environment.

#### Logging and Monitoring

The systems administrators should have the right tools for error reporting and managing issues. You should think about monitoring from the start. Distributed systems should have centralized logging. Accessing logs on ten different servers to investigate an issue is not an acceptable approach.

#### Decoupling

You should strive for the least amount of downtime during system upgrades. This means you should be able to upgrade different parts of the system independently. By making the components backward-compatible, you can update the server and the clients at different times.

By putting queues between components, you temporally decouple them. This means that, for example, the web server can still accept requests, even if the backend is down.

#### Isolate Third-Party Dependencies

You should treat systems that are outside of your control differently than components that you own. This means making your system more resilient to 3rd party failures. You can reduce the impact of an outside dependency by introducing an abstraction layer. This means that when a 3rd party system fails, you'll have fewer places to look for bugs.

### Conclusion

To work around this fallacy, you need to make your system easy to manage. DevOps, logging and monitoring can help. You also need to think about the upgrade process of your system. You cannot deploy each sprint if the upgrade requires hours of downtime. There is no one administrator so everybody should be responsible for the release process.

## 7. Transport Cost Is Zero

### The Problem

**Transport cost is not zero.**

This fallacy is related to the second fallacy, that**latency is zero**. Transporting stuff over the network has a price, in both time and resources. If the second fallacy discussed the time aspect, fallacy #7 tackles resource consumption.

There are two different sides of this fallacy:

#### The Cost of the Networking Infrastructure

The networking infrastructure has a cost. Servers, SANs, network switches, load balancers and the people who take care of this equipment — all of these cost money. If your system is deployed on-premise, then you pay this price up front. If you're using the cloud, then you pay only for what you use, but you're still paying for it.

#### The Cost of Serialization/Deserialization

The second aspect of this fallacy is the cost of transferring data between the transport level and the application level. Serialization and deserialization consume CPU time, so it costs money. If your app is deployed on-premise, this cost is somewhat hidden if you don't actively monitor resource consumption. But if your app is deployed in the cloud, this cost is painfully visible, since you're paying for what you use.

### Solutions

There isn't much you can do about the cost of infrastructure. You can only make sure that you're using it as efficiently as possible. SOAP or XML is more expensive than JSON. JSON is more expensive than binary protocols like Google's Protocol Buffers. Depending on the type of system, this can be more or less important. For example, the transport cost is much more important for apps that have to do with video streaming or VoIP.

### Conclusion

You should be mindful of the transport cost and how much serialization and deserialization your app is doing. This doesn't mean that you should optimize unless there is a need for it. You should benchmark and monitor resource consumption and decide if transport cost is a problem for you.

## 8. The Network Is Homogeneous

### The Problem

**The network is not homogenous.**

A homogenous network is a network of computers using similar configurations and the same communication protocol. Having computers with similar configurations is a hard task to achieve. For example, you have little control over what mobile devices can connect to your app. This is why it's important to focus on standard protocols.

### Solutions

You should choose standard formats in order to avoid vendor lock-in. This might mean XML, JSON or Protocol Buffers. There are plenty of options to choose from.

### Conclusion

You need to ensure that the system's components can talk with each other. Using proprietary protocols will damage your app's interoperability.

## Designing Distributed Systems Is Hard

These fallacies were published more than 20 years ago. But they still hold true today, some of them more than others. I think that many developers today know them, but the code that we write doesn't show this.

We must accept these as facts: the network is unreliable, insecure and costs money. Bandwidth is limited. The network's topology will change. Its components are not configured the same way. Being aware of these limitations will help us design better distributed systems.

**Reference:**

**https://dzone.com/articles/understanding-the-8-fallacies-of-distributed-syste**