-- is the next group of topics in 6.033 call fault tolerance.

And the goal here is to learn how to build reliable systems.

An extreme case, or at least our ideal goal, is to try to build systems that will never fail.

And what will find is that we really can't do that, but what we'll try to do is to build systems which maybe fail less often than if you built them without the principles that we're going to talk about.

So the idea is how to build reliable systems.

So in order to understand how to build reliable systems, we need to understand what makes systems unreliable.

And that has to do with understanding what faults are.

What problems occur in systems that cause systems to fail?

And you've actually seen many examples of faults already.

Informally, a fault is just some kind of a flaw or a mistake that causes a component or a module not to perform the way it's supposed to perform.

And we'll formalize this notion a little bit today as we go along.

So there are many examples of faults, several of which you've already seen.

A system could fail because it has a software fault, a bug in a piece of software, so when you run it, it doesn't work according to the way you expect.

And that causes something bad to happen.

You might have hardware faults.

You store some data on a disk.

You go back and read it.

And it isn't there.

But it's been corrupted.

And that's an example of a fault that might cause bad things to happen if you build a system that relies on a disk storing data persistently.

You might have design faults where a design fault might be something where you try to, let's say, figure out how much buffering to put into a network switch.

And you put into little buffering.

So what ends up happening is too many packets get dropped.

So you might actually just have some bad logic in there, and that causes you to design something that isn't quite going to work out.

And you might, of course, have implementation faults where you have a design, and then you implement it, and you made a mistake in how it's implemented.

And that could cause faults as well.

And another example of the kind of faults is an operational fault sometimes called a human error where a user actually does something that you didn't anticipate or was told not to do, and that caused bad things to happen.

For all of these faults, there are really two categories of faults regardless of what kind of fault it is.

The first category of faults are latent faults.

So an example of a latent fault is, let's say you have a bug in a program where instead of testing if A less than B, you test if A greater than B.

So that's a bug in a program.

But until it actually runs, until that line of code runs, this fault in the program isn't actually going to do anything bad.

It isn't going to have any adverse effect.

And therefore, this fault is an example of a latent fault.

And nothing is happening until it gets triggered.

And when it gets triggered, that latent fault might become an active fault.

Now, the problem when a latent fault becomes an active fault is that when you run that line of code, you might

have a mistake coming out at the output, which we're going to call an error.

So when an active fault is exercised, it leads to an error.

And the problem with errors is that if you're not careful about how you deal with errors, and most of what we're going to talk about is how to deal with errors, if you're not careful about how you deal with errors that leads to a failure.

So somewhat more formally, a fault is just any flaw in an underlying component or underlying subsystem that your system is using.

Now, if the fault turns out not to be exercised then.

There's no error.

There's no error that results, and there's no failure that results.

It's only when you have an active fault that you might have an error.

And when you have an error, you might have a failure.

And what we're going to try to do is to understand how to deal with these errors so that when errors occur we're going to try to hide them or mask them, or do something such that these errors don't propagate and cause failures.

So the general goal, as I mentioned before, is to build systems that don't fail.

So, in order to build systems that don't fail, there are two approaches at a 50,000 foot level.

One approach to build a system that doesn't fail is to build it out of, make sure, every system is going to be built out of components or modules.

And those modules are going to be built out of modules themselves.

One approach might be to make sure that no module ever fails, that no component that you use to build your bigger system ever fails.

And it'll turn out that for reasons that will become clear based on an understanding of the techniques we are going to employ to build systems that don't fail, it'll turn out that this is extremely expensive.

It's just not going to work out for us to make sure that our disks never fail, and memory never fails, and our

networks never fail, and so on.

It's just too expensive and nearly impossible.

So what we're going to do actually is to start with unreliable components.

And we're going to build reliable systems out of unreliable components or modules more generally.

And what this means is that the system that you build had better be tolerant of faults that these underlying components have, which is why the design of systems that don't fail or rarely fail is essentially the same as designing systems that are tolerant of faults, hence fault tolerance.

So that's the reason why we care about fault tolerance.

So let's take the example of the kinds of, just to crystallize these notions of faults and failures a little bit more.

So let's say you have a big system that has a module.

Let's call it M1. And this module uses a couple of other modules, M2 and M3. And let's say M2 uses another module, M4, where users might be an [indication?].

Or imagine this is an RPC call, for example.

And, let's say that M4 in here has some component inside M4 like a disk or something, some piece of software in M4. And, let's say that that fails.

So, it has a fault.

It gets triggered.

It becomes active, leads to an error.

It actually fails, that little component.

So when this fault becomes a failure, a couple of things could happen. M4, which is the module to which this little failure belongs, can do one of two things.

One possibility is that this fault that caused the failure gets exposed to the caller.

So, M4 hasn't managed to figure out a way to hide this failure from M2, which means that the fault propagates up.

The failure gets visible, and the fault propagates up to M2. And now, M2 actually sees the underlying component's

failure.

So the point here is that this little component fault caused a failure here which caused M4 itself to fail because M4 now couldn't hide this underlying failure, and reported something that was a failure, that was an output that didn't conform to the specification of M4 out to M2. Now, as far as M2 is concerned, all that has happened so far is that the failure of this module, M4, has shown up as a fault to M2, right, because an underlying module has failed.

It doesn't mean that M2 has failed.

It just means that M2 has now seen a fault.

And M2 now might manage to hide this fault, which would mean that M1 doesn't actually see anything.

It doesn't see the underlying fault that caused the failure at all.

But of course, if M2 now couldn't hide this or couldn't mask this failure, then it would propagate an erroneous output out to M1, an output that didn't conform to the specification of M2, leading M1 to observe this as a fault, and so on.

So, the general idea is that failures of sub-modules tend to show up as faults in the higher level module.

And our goal is to try to somehow design these systems that use lots of modules and components where at some level in the end we would like to avoid failing overall.

But inside here, we won't be able to go about making everything failure-free. I mean, there might be failures inside sub-modules. But the idea is to ensure, or try to ensure, that M1 itself, the highest level system, doesn't fail.

So let's start with a few examples.

In fact, these all examples of things that we've already seen.

And even though we haven't discussed it as such, we've seen a lot of examples of fault tolerance in the class so far.

So, for example, if you have bad synchronization code like you didn't use the locking discipline properly or didn't use any of the other synchronization primitives properly, you might have a software fault that leads to the failure of a module.

Another example that we saw when we talk about networking is when we talk about routing where the idea in here was that we talked about rat in protocols that could handle failures of links.

So, certain links could fail, leading to certain paths to not be usable.

But, the routing system managed to find other paths around the network.

And that was because there were other parts available because the network itself was built with some degree of redundancy underneath.

And the routing protocol was able to exploit that.

Another example that we saw again from networks is packet loss.

We had best effort networks that would lose packets.

And it didn't mean that your actual transfer of a file at the ends and where would miss data.

We came up with retransmissions as a mechanism to use, again, another form of redundancy where you try the same thing again to get your data through.

Another example of the failure that we saw was congestion collapse -- -- where there was too much data being sent out into the network too fast, and the network would collapse.

And our solution to this problem was really to shed load was to run the system slower than it otherwise would, by having the people sending data send data slower in order to alleviate this problem.

Another example which we saw last time was, or briefly saw last time was [the domain?] name system where the domain name servers are replicated.

So, if you couldn't reach one to resolve your domain name, you could go to another one.

And all of these, or most of these actually use the same techniques that we're going to talk about.

And all of these techniques are built around some form of redundancy on another except probably the locking thing.

But all of the others are built around some form of redundancy.

And we'll understand this more systematically today and in the next couple of classes.

So our goal here is to develop a systematic approach -- -- to building systems that are fault tolerant.

And the general approach for all fault tolerant systems is to use three techniques.

So the first one we've already seen, which is don't build a monolithic system.

Always build it around modules.

And the reason is that it will that be easier for us to isolate these modules firstly one from another.

But then, when modules fail, it will be easier for us to treat those failures as faults, and then try to hide those faults, and apply the same technique, which brings us to the second step, which is when failures occur causing errors, we need a plan for the higher level module to detect errors.

So failure results in an error.

We have to know that it's happened, which means we need techniques to detect it.

And, of course, once we detect an error we have a bunch of things we could do with it.

But ideally, if you want to prevent the failure of that system, of a system that's observed errors, you need a way to hide these errors.

The jargon for this is mask errors.

And if we do this, if we build systems that do this, then it's possible for us to build systems that can form to spec.

So the goal here is to try to make sure that systems conform to some specification.

And if things don't conform to a specification, then that's when we call it a failure.

And sometimes we play some tricks where in order to build systems that "never fail", we'll scale back the specification to actually allow for things that would in fact be considered failures, but are things that still would conform to the spec.

So we relax the specification to make sure that we could still meet the notion of a failure free system or a fault tolerant system.

And we'll see some examples of that actually in the next lecture.

And I've already mentioned, the general trick for all of these systems that we're going to study, examples that we're going to study, is to use some form of redundance.

And that's the way in which we're going to mask errors.

And almost all systems, or every system that I know of that's fault tolerant uses redundancy in some form or another.

And often it's not obvious how it uses it.

But it does actually use redundancy.

So I'm going to now give an example that will turn out to be the same example we'll use for the next three or four lectures.

And so, you may as well, you should probably get familiar with this example because we're going to see this over and over again.

It's a really simple example, but it's complicated enough that everything we want to learn about fault tolerance will be visible in this example.

So it starts with the person who wants to do a bank transaction at an ATM, or a PC, or on a computer.

You want to do a bank transaction.

And the way this works, as you probably know, is it goes over some kind of a network.

And then, if you want to do this bank transaction,, it goes to a server, which is run by your bank.

And the way this normally works is that the server has a module that uses a database system, which deals with managing your account information.

And because you don't want to forget, and the bank shouldn't forget how much money you have, there is data that's stored on disk.

And we're going to be doing things that are actions of the following form.

We're going to be asking the transfer from some account to another account some amount of money.

And now, of course, anything could fail in between.

So, for example, there could be a problem in the network.

And the network could fail.

Or the software running on the server could fail.

Or the software running this database system could crash or report bad values or something.

The disc could fail.

And do we want systematic techniques by which this transfer here or all of these calls that look a lot like transfer do the right thing?

And so, this doing the right thing is an informal way of saying meet a specification.

So, we first have to decide what we want for a specification.

That has to hold true no matter what happens, no matter what failures occur.

So one example of a specification might be to say, no matter what happens, if I invoke this and it returns, then this amount of money has to be transferred from here to here.

So that could be a specification that you might expect.

It also turns out this specification is extremely hard to meet.

And we're not even going to try to do it.

And this is the [weasel?] wording I said before about, we'll modify the specification.

So, we'll change the specification going forward for this example to mean, if this call returns, then no matter what failures occur, either a transfer has happened exactly once, or the state of the system is as if the transfer didn't even get started, OK, which is reasonable.

I mean, and then if you really care about moving the money, and you are determined that it hasn't been moved, you or some program might try it again, which actually is another form of using redundancy where you just try it over again.

And we won't understand completely why a specification that says you have to do this exactly once if it returns, why that's hard to implement, why that's hard to achieve, we'll see that in the next couple of classes.

So, for now, just realized that the specification here is it should happen exactly once or it should be as if no partial action corresponding to the [UNINTELLIGIBLE] of this transfer happened.

So the state of the system must be as if the system never saw this transfer request.

So any module could failure.

So, let's take some examples of failures in order to get some terminology that'll help us understand faults.

So one thing that could happen is that you could have a disk failure.

So the disc could just fail.

And one example of a disk failure is the disk fails and then it just stops working.

And it tells the database system that's trying to read and write data from it that it isn't working.

So if that kind of failure happens where this module here with this component just completely stops and tells the higher-level module that it stopped, that's an example of a failure.

That's called a fail stop failure.

And more generally, any module that tells the higher-level module that it just stops working without reporting anything else, no outputs, that's fail stop.

Of course, you could have disks.

And you might have failures that aren't fail stop.

You might have something where there is some kind of error checking associated with every sector on your disk.

And, disk might start reporting errors that say that this is a bad sector.

So, it doesn't fail stop, but it tells the higher level, the database system in this case that some data that's read, or some data that's been written, there's a bad sector, which means that the checksum doesn't match the data that's being read.

When you have an error like that where it doesn't stop working but it tells you that something bad is going on, that's an example of a failure.

That's called a fail fast failure.

I actually don't think these terms, that most of these terms are particularly important.

Fail stop is usually important and worth knowing, but the reason to go through these terms is more to understand that there are various kinds of failures possible.

So in one case it stops working.

In another case, it just tells you that it's not working but continues working.

It tells you that certain operations haven't been correctly done.

Now, another thing that could happen when, for example, the disc has fail stop, has fail fast is that the database system might decide that right operations, you're not allowed to write things to disk because the disk is either fail completely or is fail fast.

But it might allow actions or requests that are read only.

So, for example, it might allow users to come up to an ATM machine, and just read how much money they have from their account because it might be that there is a cache of the data that's in memory in the database.

So it might allow read-only actions, in which case the system's perform is functioning with only a subset of the actions that it's supposed to be taking.

And if that happens, that kind of failure is called a fail soft failure, where not all of the interfaces are available, but a subset of the interfaces are available and correctly working.

And the last kind of failure that could happen is that in this example, let's say that failures are occurring when there's a large number of people trying to make these requests at ATMs.

And, there is some problems that have arisen.

And somebody determines that the problem arises when there is too many people gaining access to the system at the same time.

And the system might now move to a mold where it allows only a small number of actions at a time, a small number of concurrent actions at a time, or maybe one action at a time.

So, one user can come at a time to the system, which means the systems, there has been a failure, but the way the system's dealing with it is that it determines that the failure doesn't get triggered when the load is low.

So it might function at low performance.

It still provides all of the interfaces, but just at very low performance or at lower performance.

And that kind of behavior is called failsafe.

So it's moved to a mode where it's just scaled back how much work it's willing to do, and does it at degraded

performance.

OK, so the plan now is for the rest of today, so tomorrow from the next lecture on, what we're going to do is understand algorithms for how we go about and how you build systems that actually do one or all of these in order to meet the specification that we want.

But before we do that you have to understand a little bit about models for faults.

In order to build fault tolerant systems, it's usually a good idea to understand a little bit more quantitatively models or faults that occur in systems.

And primarily, this discussion is going to be focused on hardware faults because most people don't understand how software faults are to be modeled.

But since all our systems are going to be built on hardware, for example discs are going to be really, really common.

Our network links are going to be common.

And all of those conform nicely to models.

It's worth understanding how that works.

So, for example, a disk manufacturer might report that the error rate of undetected errors, so disks usually have a fair amount of error detection in them.

But, they might report that the error rate of undetected errors is, say, ten to the minus 12 or ten to the minus 13. And that number looks really small.

That says that out of that many bits, maybe one bit is corrupted, and you can't detect it.

But, you have to realize that given modern workloads, for example take Google as an example that you saw from last recitation, the amount of data that's being stored in the system like that or in the world in general is so huge that a ten to the minus 13th error rate means that you're probably seeing some bad data and file that you can never fix or never detect every couple of days.

Network people would tell you that fiber optic links have an error rate of one error in ten to the 12th. But you have to realize that these links are sending so many gigabits per second that one error in ten to the 12th means something like there's an error that you can't detect maybe every couple of hours.

What that really means is that at the higher layers, you need to do more work in order to make sure that your data is protected because you can't actually rely on the fact that your underlying components have these amazingly low numbers because there's so much data going on, being sent or being stored on these systems that you need to have other techniques at a higher layer to protect, if you really care about the integrity of your data.

In addition to these raw numbers, there's two or three other metrics that people use to understand faults and failures.

The first one is the number of tolerated failures.

So, for example, if you build a system to store data and you're worried about discs failing or discs reporting at [earnest?] values, you might replicate that data across many, many discs.

And then when you design your system, one of the things you would want to analyze and report is the number of tolerated failures of discs.

So, for example, if you build a system out of seven discs, you might say that you can handle up to two failed disks, or simply like that, depending on how you've designed your system.

And that's usually a good thing to report because then people who use your system can know how to provision or engineer your system.

The second metric which we're going to spend a few more minutes on is something called the mean time to failure.

And what this says is it takes a model where you have a system that starts at time zero, and it's running fine.

And then at some point in time, it fails.

And then, when it fails, that error is made known to an operator.

Or it's made known to some higher level that has a plan to work around it to repair this failure.

And then, once the failure gets repaired, it takes some time for the failure to get repaired.

And once it's repaired it starts running again.

And then it fails at some other point in the future.

And when it goes through the cycle of failures and repairs, you end up with a timeline that looks like this.

So you might start at time zero.

And the system is working fine.

And then there is a failure that happens here.

And then the system is down for a certain period of time.

And then somebody repairs the system, and then it continues to work.

And then it fails again, and so on.

And so, for each of the durations of time that the system is working, let's assume it's starting at zero, each of these defines a period of time that I'm going to call TTF or time to fail.

OK, so this is the first time to fail interval.

This is the second time to fail This is the second time to fail interval, time to repair, and this is the third time to fail interval, and so on.

And analogously in between here, I could define these time to repair intervals, TTR1, TTR2, and so on.

So, the mean time to failure is just the mean of these values.

There's some duration of time.

You're like, three hours the system worked, and then it crashed.

That's TTF1. And then, somebody to repair it worked now for six hours.

That's TTF2, and so on.

If you run your system for a long enough period of time like a disk or anything else, and then you observe all these time to fail samples, and take the mean of that, that tells you a mean time to failure.

The reason this is interesting is that you could run your system for a really long period of time, and build up a metric called availability.

So, for example, if you're running a website, and the way this website works is it runs for a while and then every once in awhile it crashes.

So its network crashes and people can't get to you.

So you could run this for months or years on end, and then observe these values.

You could run this every month.

You could decide what availability is, and decide if it's good enough or if you want to make it higher or lower.

So you could now define your availability to be the fraction of time that your system is up and running.

And the fraction of time that the system is up and running is the fraction of time on this timeline that you have this kind of shaded thing.

OK, so that's just equal to the sum of all the time to failure numbers divided by the total time.

And the total time is just the sum of all the TTF's and the TTR's.

OK, and that's what availability means is the fraction of time that your system is available is up.

Now, if you divide both the top and the bottom by N, this number works out to be the mean time to failure divided by the mean time to failure plus the mean time to repair.

So this is a useful notion because now it tells you that you can [point?] your system for a very long period of time, and build up a mean estimate, mean values of the time to failure and the time to repair, and just come up with the notion of what the availability of the system is.

And then, decide based on whether it's high enough or not whether you want to improve some aspect of the system and whether it's worth doing.

So it turns out this mean time to failure, and therefore availability is related for components to a notion called the failure rate.

So let me define the failure rate.

So the failure rate is defined, it's also called a hazard function.

That's what people use the term H of T, the hazard rate.

That's defined as the probability that you have a failure of a system or a component in time, T to T plus delta T, given that it's working, we'll say, OK, at time T, OK?

So it's a conditional probability.

It's the probability that you'll fail in the next time instant given that it's correctly working and has been correctly working.

It's correctly working at time T.

So, if you look at this for a disk, most discs look like the picture shown up here.

This is also called the bathtub curve because it looks like a bathtub.

What you see at the left end here are new discs.

So, the X axis here shows time.

I guess it's a little shifted below.

You can't read some stuff that's written at the bottom.

But the X axis shows time, and the Y axis shows the failure rate.

So, when you take a new component like a new light bulb or a new disc or anything new, there is a pretty high chance that it'll actually fail because manufacturers, when they sell you stuff, don't actually sell you things without actually burning them in first.

So for semiconductors, that's also called yield.

They make a whole number of chips, and then they're burning a few, and then they only give you the rest.

And the fraction that survives the burning is also called the yield.

So what you see on our left, the colorful term for that is infant mortality.

So it's things that die when they are really, really young.

And then, once you get past the early mortality, you end up with a flat failure, a conditional probability for failure.

And what this says is that, and I'll get to this and a little bit.

But what this says is that once you are in the flat region, it says that the probability of failure is essentially independent of what's happened in the past.

And then you stay here for a while.

And then if the system has been operating like a disk has been operating for awhile, let's say three years or five years typically for discs, then the probability of failure starts going up again because that's usually due to wear and tear, which for hardware components is certainly the case.

There are a couple of interesting things about this curve that you should realize, particularly when you read specifications for things like discs.

Disc manufacturers will report a number, like the mean time to failure number.

And the mean time to failure number that the report might usually, I mean for discs might be 200,000 hours or 300,000 hours.

I mean, that's a really long period of time.

That's 30 years.

So when you look at a number like that, you have to ask whether what it means is that discs really survive 30 years.

And anybody who is on the computer knows, you know, most discs don't survive 30 years.

So they are actually reporting one over the reciprocal of this thing at the flat region of the curve because this conditional failure probability rate, at this operation time when the only reason things fail is completely random failures not related to wear and tear.

So when disc manufacturers report a mean time to failure number, they are actually reporting something that that's the time that you're disc is likely to work.

What that number really says is that during the period of time that the disc is normally working, the probability of a random failure is one over the mean time to failure.

That's what it really says.

So the other number that they also report, often in smaller print, is it the expected operational lifetime.

And that's usually something like three years or four years or five years, whatever it is they report.

And that's where this thing starts going up, and beyond a point where the probability of failures above some threshold, they report that as the expected operational lifetime.

Now, for software, this curve doesn't actually apply, or at least nobody really knows what the curve is for software.

What is true for software, though, is infant mortality, things were the conditional probability of failure is high for new software, which is why you are sort of well advised, the moment the new upgrade of something comes around, most people who are prudent wait a little bit to just make sure all the bugs are out, and things get a little bit stable.

So they wait a few months.

You are always a couple of revisions behind.

So I do believe that for software, the left side of the curve holds.

It's totally unclear that there is a flat region, and it's totally unclear that things start rising again with age.

So the reason for this curve, being the way it is, is a lot of this is based on the fact that things are mechanical and have wear and tear.

But the motivation for this kind of curve actually comes from demographics and from human lifespans.

So this is a picture that I got from, it's a website called mortality.org, which is a research project run by demographers.

And they have amazing data.

There's way more data available but human life expectancy and demographics than anything about software.

What this shows here is actually the same bathtub curve as in the previous chart.

It just doesn't look like that because the Y axis is on a log scale.

So given that it's rising linearly between 0.001 and 0.01, on a linear scale that looks essentially flat.

So human beings for the probability of death, at a certain time, given that you are alive at a certain time, that follows this curve here, essentially a bathtub curve. At the left hand, of course, there is infant mortality.

I think I pulled the data down.

I think I pulled the data down.

This is from an article that appeared where the data for the US population 1999. It starts off again with infant mortality.

And then it's flat for a while.

Then it rises up.

Now, there's a lot of controversy, it turns out, for whether the bathtub curve at the right end holds for human beings or not.

And, some people believe it does, and some people believe it doesn't.

But the point here is that for human beings anyway, the rule of thumb that insurance companies use for determining insurance premiums is that the log of the death rate, the log of the probability of dying in a certain age grows linearly with the time that somebody has been alive.

And that's what this graph shows, that on [large?] scale on the Y axis, you have a line.

And that's what they use for determining insurance premiums.

OK, so the reason this bathtub curve is actually useful is, so if you go back, let's go back here.

The reason both these numbers are useful, the flat portion of the bathtub curve and the expected operational lifetime is the following.

It's not like this flat portion of the curve where the disc manufacturer reports the mean time to failure.

That's 30 years.

It's not like that's useless even though you're disc only might run for three to four years.

The reason is that if you have a project, if you have a system where you are willing to upgrade your disk every three years where you've budgeted for upgrading your discs every said three years, then you might be better off buying a disk whose expected lifetime is only five years but whose flat portion is really low.

So in particular, if you're given to discs, one of which has a curve that looks like that, and another that has a curve that looks like that, and let's say this is five years, and this is three years.

If you're building a system and you've budgeted for upgrading your discs every four years, then you're probably better off using the thing with the lower value of mean time to failure because its expected lifetime is longer.

But if you're willing to upgrade your discs every two years or one year, then you might be better off with this thing here with the lower meantime to failure, even though its expected operational lifetime is smaller.

So both of these numbers are actually meaningful, and it depends a lot on how you're planning to use it.

I mean, it's a lot like spare tires on your car.

I mean, the spare tire was run perfectly fine as long as you don't exceed 100 miles.

And the moment you exceed 100 miles, then you don't want to use it at all.

And it might be a lot cheaper to build the spare tire that runs just 100 miles because the users, you are guaranteed that you will get to a repair shop is in 100 miles.

It's the same concept.

OK. So one of the things that we can define, once we have this condition of failure rate is the reliability of the system.

We'll define that as the probability, R of T, is the probability that the system's working at time T, given that it was working at time zero, or more generally assuming that everything is always working at time zero, it's the probability that you're OK at time T.

And it turns out that for components in the flat region of this curve, H of T, the conditional failure rate is a constant, on systems that satisfy that, and would satisfy the property that the actual unconditional failure rate is a [memory-less?] process where the probability of failure doesn't depend on how long the system's been running.

It turns out that for the systems that satisfy those conditions, which apparently discs do in the operation when they're actually not at the right edge of the curve, which discs do, the reliability, this function goes as the very nice, simple function, which is an exponential decaying function, E to the minus T over MTTF.

And this is under two conditions.

H of T has to be flat, and the unconditional failure rate has to be something that doesn't depend on how long the system's been running.

And for those systems, it's not hard to show that your reliability is just an exponential decaying function, which means you can do a lot of things like predict how long the system is likely to be running, and so on.

And that will tell you when to upgrade things.

OK, so given all of this stuff, we now want techniques to cope with failures, cope with faults.

And that's what we're going to be looking at for the next few lectures, let's take one simple example of a system first.

And like I said before, all of these systems use redundancy in some form.

So the disk fails at a certain rate.

Just put in multiple disks, replicate the data across them, and then hope that things survive.

So the first kind of redundancy that you might have in the example that I just talked about, spatial redundancy, where the idea is that you have multiple copies of the same thing, and the games we're going to play all have to do with how we're going to manage all these copies.

And actually, this will turn out to be quite complicated.

We'll use these special copies in a number of different ways.

In some examples, we'll apply error correcting codes to make copies of the data or use other codes to replicate the data.

We might replicate data and make copies of data in the form of logs which keep track of, you know, you run an operation.

You store some results.

But at the same time, before you store those results, you also store something in a log [surf?], the original data went away; your log can tell you what to do.

Or you might just do plain and simple copies followed by voting.

So the idea is that you have multiple copies of something, and then you write to all of them.

In the simplest of schemes, you might write to all of them, and then when you want to read something, you read from all of them.

And then just what?

And go with the majority.

So intuitively that can tolerate a certain number of failures.

And all of these approaches have been used.

And people will continue to build systems along all of these ideas.

But in addition, we're also going to look at temporal redundancy.

And the idea here is try it again.

So this is different from copies.

What it says is you try something.

If it doesn't work and you determine that it doesn't work, try it again.

So retry is an example of temporal tricks.

But it will turn out will also use not just moving forward and retrying something that we know should be retried, we'll also use the trick of undoing things that we have done.

So we'll move both directions on the time axis.

We'll retry stuff, but at the same time we'll also undo things because sometimes things have happened that shouldn't have happened.

Things went half way.

And we really want to back things out.

And we were going to use both of these techniques.

So one example of spatial redundancy is a voting scheme.

And you can apply this to many different kinds of systems.

But let's just apply it to a simple example of, there is data stored in multiple occasions.

And then whenever data is written, it's written to all of them.

And then when you read it, you read from all them, and then you vote.

And in a simple model where these components are fail stop, which means that they fail; they just fail.

Excuse me, on a simple model where things are not fail stop or fail fast, but just report back this data to you, so

you are voting on it, and these results come back.

You've written something in them and when you read things back, arbitrary values might get returned if there's a failure.

And if there's no failure here, correct values get returned.

Then as long as two of these copies are correctly working, or two of these versions are correctly working, then the vote will actually return to you at the correct output.

And that's the idea behind voting.

So if the reliability of each of these components is some R, that's the probability that the system's working at time T according to that definition of reliability.

Then, under the assumption that these are completely independent of each other, which is a big assumption, particularly for software.

But it might be a reasonable assumption for something like a disk, under the assumption that these are completely independent, then you could write out the reliability of this three-voting scheme of this thing where you are voting on three outputs.

But you know that the system is correctly working if any two of these are correctly working.

So that happens under two conditions.

Firstly, all three are correctly working, right?

Or, some two of the three are correctly working.

And, there's three ways in which you could choose some two of the three.

And, one of them is wrongly working.

And it turns out that this number actually is very, very large, much larger than R, when R is close to one.

And, in general, this is bigger than R when each of the components has high enough reliability, namely, bigger than half.

And so, let's say that each of these components has a reliability of 95%. If you work this number out, it turns out to be a pretty big number, much higher than 95%, much closer to one.

And, of course, this kind of voting is a bad idea if the reliability of these components is really low.

I mean, if it's below one half, then chances are that you're more likely the two of them are just [wrong?], and you agree on that result.

And it turns out to reduce the reliability of the system.

Now, in general, you might think that you can build systems out of this basic voting idea, and for various reasons it turns out that this idea has limited applicability for the kinds of things we want to do.

And a lot of that stems from the fact that these are not, in general, in computer systems.

It's very hard to design components that are completely independent of each other.

It might work out OK for certain hardware components where you might do this voting or other forms of spatial redundancy that gives you these impressive reliability numbers.

But for software, this independent assumption turns out to be really hard to meet.

And there is an approach to building software like this.

It's called N version programming.

And it's still a topic of research where people are trying to build software systems out of voting.

But you have to pay a lot of attention and care to make sure that these software components that are doing the same function are actually independent, maybe written by different people running on different operating systems, and so on.

And that turns out to be a pretty expensive undertaking.

It's still sometimes necessary if you want to build something highly reliable.

But because of its cost it's not something that is the sort of cookie-cutter technique for achieving highly reliable software systems.

And so what we're going to see starting for next time is a somewhat different approach for achieving software reliability that doesn't rely on voting, which won't actually achieve the same degree of reliability as these kinds of systems, but will achieve a different kind of reliability that we'll talk about starting from next time.