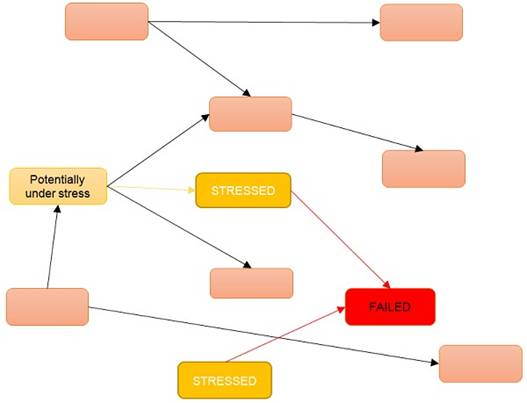
**Microservices:**

Sometimes things happen around a microservice that may disturb the normal operation of the microservice. We say that the microservice is under stress in such situations. There are many sources of stress. To name a few, a microservice may be under stress because:

* One of the machines in the cluster its data store runs on has crashed
* It has lots network connectivity to one of its collaborators
* It is receiving unusually high amounts of traffic
* One of its collaborators is down

In all of these situations, the microservice under stress cannot continue to operate the way it normally does. That doesn't mean that it's down, only that it must cope with the situation.

When one microservice fails, its collaborators are put under stress and are also at risk of failing. While the microservice is failing, its collaborators will not be able to query, send commands or poll events from the failing microservice. As illustrated in figure 3, if this makes the collaborators fail, even more microservices are at risk of failing. At this point, the failure has started propagating through the system of microservices. Such a situation can quickly escalate from one microservice failing to lot of microservices failing.

  
**Figure 3:** If the microservice marked FAILED is failing, so is the communication with it. That means that the microservices at the other end of those communications are under stress. If the stressed microservices fail due to the stress, the microservices communicating with them are put under stress. In that situation, the failure in the failed microservice has propagated to several other microservices.

Some examples of how we can stop failures propagating are:

* When one microservice tries to send a command to another microservice, which happens to be failing at the time, that request will fail. If the sender simply fails as well, we get the situation illustrated in figure 3 where the failures propagate back through the system. To stop the propagation, the sender might act as if the command succeeded, but actually store the command into a list of failed commands. The sending microservice can periodically go through the list of failed commands and try to send them again. This is not possible in all situations, because the command may need to be handled here and now, but when this approach is possible it stops the failure in one microservice from propagating.
* When one microservice queries another one that's failing, the caller could use a cached response. In case the caller has a stale response in the cache, but a query for a fresh response fails, it might decide to use the stale response anyway. Again, this is not something that will be possible in all situations, but when it is, the failure will not propagate.
* An *API Gateway* that is stressed because of high amounts of traffic from a certain client can throttle that client by not responding to more than a certain number of requests per second from that client. Notice that the client may be sending an unusually high amount of requests because it is somehow failing internally. When throttled, the client will get a degraded experience, but will still get some responses. Without the throttling, the *API Gateway* might become slow for all clients or it might fail completely. Moreover, since the *API Gateway* collaborates with other microservices, handling all the incoming requests would push the stress of those requests onto other microservices too. Again, the throttling stops the failure in the client from propagating further into the system to other microservices.