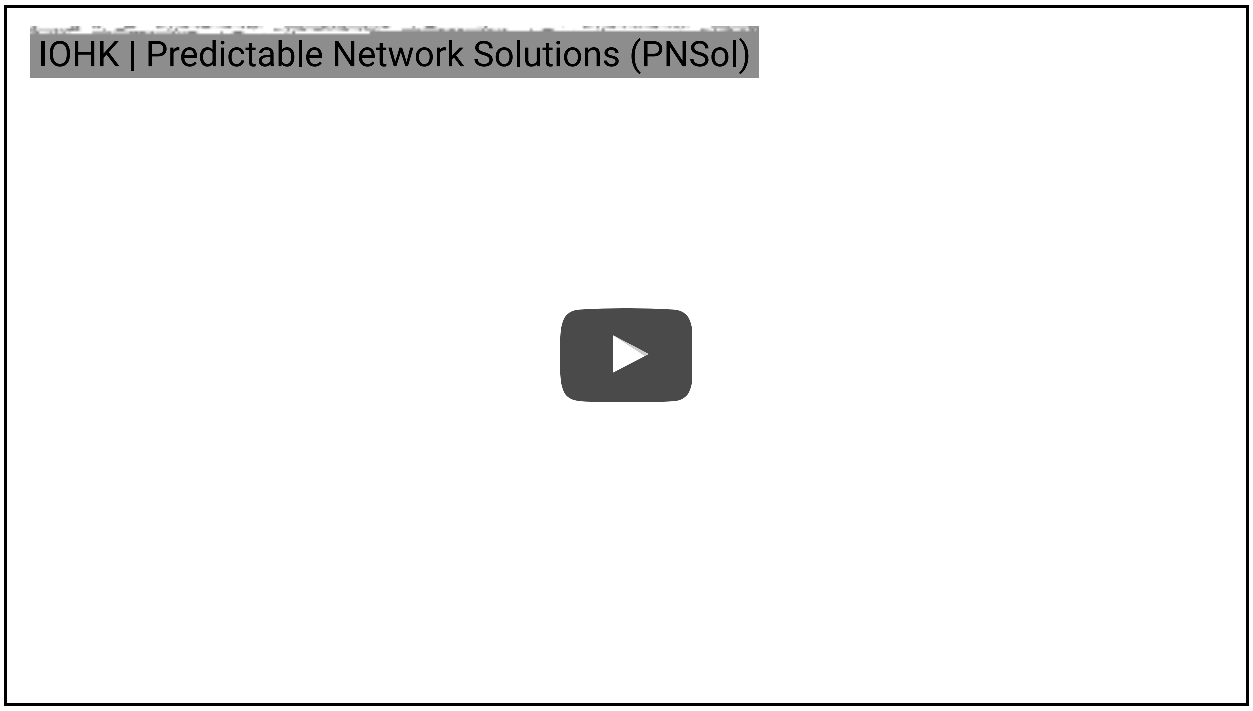
Cardano: Resilient and Scalable by Design

System performance engineering so DevOps may soundly sleep

AUGUST 03, 2017 6 MIN READ



What we expect from traditional providers of financial services such as banks is both security (my money is safe) and responsiveness (I can move my money in and out at will in a timely fashion). The days when banks delivered such services using legions of clerks writing in double-entry ledgers are long gone – nowadays it's all done by software, and so the security and performance of such systems is critical to a bank's reputation. Banks invest heavily in their computing and network infrastructure and personnel to mitigate this risk (for example, we happen to know the Head of Unix System Engineering at a major international bank is paid a LOT more than any of us!). Customer expectations are high, tolerance of poor performance is low, and it is the poor DevOps who end up dealing with the consequences of any deficiencies and emergent instabilities.

Another advantage that traditional banks have is periods when they aren't expected to be fully operational –

after local markets close, for example, and during public holidays. As global infrastructure, Cardano needs to run both continuously and indefinitely. Its resulting performance needs to be acceptable even in the presence of hardware or software failures and cyber-attacks – and it must do all this without constant maintenance from a large DevOps team.

This requires the application of the emerging discipline of distributed systems performance engineering to anticipate and mitigate the issues associated with long-term, continuous and scalable operation. This combines failure-mode effects analysis with stochastics (which uses probability and randomness over real-time) to model the impact of both resource-sharing (for example in packet networks and virtualised infrastructure) and the possibility of failures and exceptions. Of course, it's natural to ask: if the software correctly implements the specification, how can there be failures? What have we done wrong? The answer is that we haven't done anything wrong, it's just that we're not operating in a closed environment. The real world is an open environment, many elements of which are not under our control. Messages between components of Cardano may be lost or corrupted, either by accident or malicious intent; VMs running such components may crash or be starved of resources; and DoS attacks may exhaust resources. Even if our code is perfect, the world in which it runs is not.

Performance engineering can be used in a post-hoc way to assess the expected performance and resource consumption of an existing system, but for Cardano we'll use it to help guide design decisions in the re-implementation of the network layer. In a previous blog, Duncan Coutts of Well-Typed, and Cardano's Director of Engineering, talked about how formal methods can help to ensure that a design decision doesn't break the top-level specification of what the system should (and should not) do; what performance engineering adds is an assessment of whether such a design decision moves us closer to (or further away from) meeting the resilience, performance and scalability targets for the eventual deployment.

Current state of the art

With the exception of "hard real time" systems such as anti-lock brake controllers, it's rare to see performance, resilience and robustness treated as first-class citizens in the software development lifecycle (SDLC). Even where such properties are considered, this typically occurs late in the SDLC. Performance, in particular, is regarded as something that will "emerge" after the design, and much, if not all, of the implementation has been done. Although robustness, resilience and performance are closely linked, let's focus on performance, as this is the most widely misunderstood.

In the academic world, system steady-state performance has been widely studied using queueing theory. Approaches tend to take a resource-centric view of the system, focusing on the utilisation/idleness of individual components. Where job/customer performance is considered (such as in mean-value analysis or Jackson/BCMP networks) it is in the context of "steady-state" and "averages". Thus these methods cannot deliver metrics such as the distribution of the system's response time to a stimulus, or the probability that such a response will occur within a specified time.

Meanwhile, in today's customer experience-centric, performance-critical service-delivery environments, such metrics are essential. An end customer doesn't care how efficient the system is, only how long it takes to process her transaction; and an acceptable average does not compensate for the disappointment of a particular transaction taking a hundred times too long!

This has led academic research into the characterisation of "passage-times", i.e. the time taken for a system to follow a particular path to a state, that path being characteristic of an outcome. Such a style of analysis has been combined with stochastic/probabilistic algebras to generate tools that can be applied in the SDLC, such as PEPA and PRISM. These are retrospective validation tools, operating on fully specified systems, that will give probabilistic measures of outcomes for certain classes of system under steady-state assumptions.

However, constructing a large-scale system such as Cardano is expensive, and no-one wants to iterate large parts of the design just because the required performance is not achieved and/or the resources consumed are uneconomic. Mitigating that risk requires an approach that supports both prospective and retrospective validation and verification. It needs to be able to capture performance requirements, to validate them, to construct performance properties/invariants that "witness" those requirements, and to support the reification and abstraction of such properties/invariants throughout the SDLC. In other words, the analysis approach needs to be composable at all points in the SDLC, a property which all of the other approaches above lack with respect to performance.

A composable approach

Composability is the key to managing complexity in the SDLC. The principle of composability is as follows: the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. For composable properties, what is "true" about small subsystems (e.g. their timeliness, their resource consumption) tells us what is "true" about their (appropriately constructed) combination. Conversely, it means that there is an invariant that must hold (e.g. timeliness, aspects of functional correctness) over the reified components of the system.

This is the same as checking functional correctness by breaking down a top-level specification into a number of component specifications and proving that the combination meets the top-level spec.

Engagement with the general notion of composability, and the associated improvement in productivity, can be seen in the increasing tendency of leading ICT practitioners (e.g. Google, Facebook, WhatsApp, leading banks' real-time trading systems – and of course, Cardano) to use functional/declarative programming approaches such as Haskell for their key systems. Such approaches are improving the verification and validation (V&V) of functional aspects of software systems; composable performance engineering represents a similar step-change in the V&V of the "non-functional" aspects of performance and resource consumption.

PNSol has developed a framework around a composable measure of performance that we call "ΔQ". This enables a new development process that is composable with respect to both performance hazards (i.e. time to complete and probability of non-completion/divergence) and aspects of resource consumption (e.g. CPU time, network/interconnect capacity). PNSol represent the operational semantics with a stochastic process algebra (using a combination of improper random variables and serial-parallel flow graphs), to capture both communication and computational behaviour. This approach has a supporting software library/API that PNSol has been using for more than 10 years in consultancy engagements, which supports both symbolic and concrete representations of the metrics of interest, helping to capture design and operational uncertainties as part of the SDLC.

This approach also helps to pinpoint performance sensitivities, i.e. to reveal which parameters have the most impact on the eventual system performance. The DevOps team can then know what to measure, track, and trend in order to have early warning of performance or resource consumption problems, and hence can get some sleep from time to time!

Applying the ΔQ Framework

To apply this in practice we need to first establish some "quantifiable intent", that is to say, to set bounds on the performance of some observable behaviour of the system. An observable is something that starts and finishes; in Cardano we might think about submitting a transaction and seeing it embedded in the blockchain, although a simpler and more familiar example would be clicking a button on a web page and getting some response. The quantified intent for that web server response might be something like: 50% of the time it should take less than 2s; 95% of the time it should take less than 5s; 99.9% of the time it should take less than 20s. Note that we allow the possibility that it might fail altogether – technically this means we represent the observable with an “improper random variable” – which is very important for dealing with the real world, in which things can (and do) fail. Taking proper account of this allows us to design systems that degrade gracefully rather than collapsing apparently arbitrarily. The next step is to extract from the design what other observables the initial one depends on, typically transfers of information across a network and computations using that information (each of which also consumes some resources). Given the way these observables are causally related (called the serial-parallel flow graph, SPFG), the ΔQ framework allows us to combine their performance distributions to calculate the resulting distribution for the original observable, and to check whether it meets the original intent (here is a worked example). If it doesn’t meet this intent, we may need to tighten the distributions for some of the component observables, or change the design to alter the SPFG. Note that we can either apply this approach top-down (as a set of performance “budgets”) or bottom-up (some elements such as network delays may not be changeable), or use a combination of the two. We can also treat a whole subsystem as delivering a single observable, and then break the delivery of that observable into its constituent parts, thus iterating the whole process – this makes the approach composable, as discussed above.

At the same time we can add up the distributions of resource consumption to obtain not merely averages but also probabilities of thresholds being exceeded. Once the relationship between delivered performance and resource consumption is properly modelled, it is straightforward to address issues of scalability, exception/failure propagation and/or mitigation, and the impact of correlations in demand (discussed in more detail here).

Applying this to something as complex as Cardano-SL will be a challenging project, but will enable us to address the issues of robustness, resilience and performance with our eyes wide open – resulting in an economic and appropriately scaled solution.