
2.1 Introduction

2.1.1 The Technical and Social Origins of the Cloud

Cloud computing is a new term, but not really such a new idea. What we call cloud computing today can be traced to the 1990s, when researchers at Xerox PARC published papers on a new idea that they called *ubiquitous computing*: a world in which computing devices would surround us, and in which we would rely upon them as casually as electric lights or tapwater.

At the time, many sober-minded computing professionals viewed this idea as being a bit over the top: in those days, *distributed computing* was still a relatively new phenomenon, and the earlier networked but relatively loosely connected computing systems were only just becoming common. It did not help that some of the early researchers in the field came across as being a bit quirky. For example, one fellow decided to capture all the events in his life digitally; he went about with cameras strapped to his head, microphones, location tracking devices, etc. These were the days when cameras and microphones and other networked devices were all big bulky things, and the photos of this skinny researcher with all those components strapped on did not exactly inspire imitation. Newspaper editorials debated the pros and cons of building systems that by their very nature intrude into what normally had been private interactions. And many of the core technologies did not even work very well. Ubiquitous computing seemed very unrealistic and over-ambitious, and very unlikely to become a reality anytime soon.

Yet however fanciful the Xerox vision may have seemed in its heyday, cloud computing has to be seen as a realization of that vision and indeed, one that goes far beyond what the Xerox research team anticipated. It turns out that your cell phone, buried in your pocket or in your purse, does track your location rather accurately, unless you disable that feature. And even if you disable GPS tracking, the phone may still know your location to very high accuracy simply by virtue of the wireless networks it can sense in the vicinity: there are growing numbers of high-resolution maps that identify the locations of wireless basestations; a mobile device can often situate itself to an accuracy of several meters by triangulating with respect to the

available wireless systems (even ones that are secured and hence not really accessible). A telephone's microphone is not supposed to be listening to you unless you are making calls, but countless movies revolve around the idea that a spy organization (or even an obsessive ex-boyfriend) might find a way to turn the phone on covertly, and apparently this is not even all that hard to do. Few people videotape their lives as they move around, but crowded public places in the UK often have video cameras trained on the crowd, and there are more and more in the United States as well.

We are putting all kinds of information online; including data of kinds that the early Xerox Parc researchers could never have imagined. For example, who could have anticipated the success of social networking sites such as Facebook and Twitter and YouTube? Using the cloud as an intermediary, people are sharing ideas and experiences and photos and videos and just about everything else in a casual, routine way; maintaining close contact with friends and family nearly continuously; and trusting our the cloud to recommend the best prices on product, to offer hints about good local restaurants, and even to set up impromptu dates with people in the vicinity who might be fun to meet. As we move about and interact, the devices around us collect data and those data enter the cloud, often being stored and indexed for later use. Not much is ever really erased, in part because this can be hard to do. Consider email: suppose a friend sends you a private email, and you read it and then delete it. How sure can you be that the data are gone? If an email-provider erases your email, does this mean it cannot also offer a recovery feature to recover precious emails that might accidentally be erased, or would that violate the policy? The question makes sense because many cloud-hosted email systems do have ways to recover deleted emails. But if deleted emails are recoverable, in what sense were they deleted? What if some index was computed and a deleted email somehow lingers within the computed result: must we recompute every such index? Again, a sensible question, because indexing occurs all the time. And what if the email provider has a backup technology in place: does erasing the email also erase the backups? Probably not: backups are typically "write once" data.

Realistically, it makes sense to just assume that *any* data that find their way online have a good chance of ending up indexed, filed away, and retained (perhaps in a derived form) for use in enhancing your web search experience and to optimize advertising placement. This is not limited to data we deliberately upload; it includes data about credit card purchases, telephone calls, the car we drive and how fast we drive it, speeding and parking tickets, common destinations and routes. It includes angry letters you exchanged with your "ex" in the period before the breakup. The cloud tracks home ownership and mortgage data, and property value trends. Any kind of public records, insurance records, marriages, divorces, arrests: it all gets collected, correlated, stored; entire companies have sprung up to play precisely this role.

Cloud providers make their money by matching your queries to the right web sites, placing advertising likely to appeal to you personally, and trying to "understand" who you are and what motivates you, so as to shape your experience positively. These goals motivate them to capture more and more information. Obviously,

some limits do apply (medical records, for example, are covered by HiPPA regulations in the United States, and other countries, such as Israel, have very strong privacy rules around digital data). Yet up to the limits allowed by the law, you can be quite sure that cloud computing companies are doing anything technically feasible that might offer even the slightest edge in the battle for screen space on your mobile device and user interest. In some cases, by accepting a user agreement, you agree to waive your rights under these kinds of data collection laws. Yet many people click to accept without reading such agreements (and only some people have the legal experience to understand them, in any case).

Even in the early days, the Xerox ubiquitous computing crowd worried about the ways that computing systems capable of capturing huge amounts of data might transform society itself, notably by eroding privacy and, in the more dystopian visions, creating a kind of smothering Big Brother society. Today, for all the fretting about privacy, most cloud users feel liberated, not oppressed by these technical trends. If someone were to assert that the cloud is becoming Big Brother, they would be viewed as being especially paranoid. There isn't even one cloud: the cloud is a world within which multiple companies compete, with no clear single winner. Moreover, those companies need to maintain customer trust. Google's "Don't be Evil" corporate motto is just one of the ways that these companies constantly remind their developers and employees of the central importance of being trusted by the user. And they are right to put trust front and center: Every major cloud company understands how easy it can be to fall from dizzying success to utter failure. Serious invasions of privacy could wipe out a major player overnight.

Of course the cloud does cut both ways, but the darker side has not been all that visible in Western society. There is no question that politicians and other public figures are finding the cloud troublesome; it seems very likely that at least some of the recent spate of "outings" of bad behavior originated with politically motivated hackers who broke into email or Twitter accounts, turned up damning information, and then found ways to arrange to leak that information to the public. Yet there has not been much outrage at the idea this might be going on: the public seems to have a real appetite for tawdry news and clearly enjoys watching the rich and famous taken down a notch or two.

Perhaps a bit more of a concern is the degree to which the cloud has simultaneously been a liberating technology in repressive countries (for example, making it easier to organize protests) but also a tool for those same governments to spy on their citizens and to identify signs of dissent, thus enabling further waves of repression. This was very visible during the so-called Arab Spring, when protesters in Iran and Egypt used Twitter and Facebook to orchestrate demonstrations, but then saw their leaders rounded up by the military a few weeks or months later, to be forcefully reminded about who was really in charge. For those working in cloud computing, the societal penetration of their technologies represent a complex and paradoxical phenomenon: leaders within the community are often proud to talk about the cloud as a powerful vehicle for individual expression, and yet how could they not also feel anxious when they see these same technologies helping the military or other repressive forces identify the most troublesome of the protesters. It is not at all clear how

this will play out, and how history will view cloud technologies and their roles in repressive societies, over the long term.

This text will not have much more to say about those kinds of issue (readers wishing to learn more might read Laurence Lessig's work on the topic (Lessig 1999)), but it is interesting to think about how the cloud has transformed our contemporary notion of privacy. In 1890, Justices Warren and Brandeis famously defined privacy as the "right to be left alone" and ultimately authored a Supreme Court decision that firmly recognized the Constitutional basis of this right. How quickly the very notion has been confused and eroded by technology, and by the collective behavior of the world's email, Twitter and Facebook users! How could one really be left alone today, short of moving to a cabin in the woods and never touching technology at all (and even then, would not the cloud preserve information about your life up to that moment, and your ownership of that cabin (perhaps even a satellite photo), and maybe even your purchases in the local store)? Professor Lessig is often quoted for his observation that technology often gets far ahead of the law, but here we see a situation in which technology has actually gotten far ahead of our societal norms, literally reshaping the way we live.

But let us return to the early days of the cloud, this time with a more technical focus. When the Xerox Parc group first conceived of ubiquitous computing, most systems were personal ones. These personal computing systems were highly autonomous: they had their own operating systems, private copies of whatever applications you wanted to run and were prepared to pay for, and used the network mostly to exchange email and data files.

The term networking was used as a catchall covering the computer network itself together with those limited kinds of network application: email, early versions of blogs, file transfer (it evolved into the Internet, but the Xerox work on ubiquitous computing actually predated what we would call the Internet today). This book will try to use networking as a term focused on connectivity that enables these kinds of application, as distinct from the way we will use the term *distributed computing*, which for us is concerned with collections of computers, connected to one-another by networks, that collaborate to jointly carry out tasks. That is, for our purposes here, a networked computing system is a machine with its own roles and objectives, which obtains data from servers (with their own roles and objectives) via network connection. Distributed computing, in contrast, is concerned with what teams of collaborating computers can jointly accomplish. The distinction is analogous to shopping as an individual (this would be the networked case), versus dividing up the work and shopping in a collaborative way with friends (the distributed case). In both cases we interact with stores (services), but in the former case the client is on his or her own. In the latter one, the clients talk to one-another and share the job.

In the earliest days of cloud computing, networking was the bread and butter from which most applications were created. Distributed computing was the nec-plus-ultra of systems at the time. As Leslie Lamport¹, a researcher at Digital Equipment Com-

¹Lamport's comment about distributed system may have been somewhat pessimistic, but as we will see later in this book, he was in the midst of developing what we now think of as the theo-

pany's SRC laboratory put it, distributed systems were ones in which your personal computer could become unusable because of a failure in some other computer that you had never heard of, playing a role that you were unaware that you depended upon: not really the most positive or flattering way to characterize the technology. In contrast, if the network was down, you could not read your email, but could still use your computer to write software (or play Tetris).

Although some Xerox Parc researchers expected that Lamport's style of distributed system would turn out to be the key to ubiquitous computing, most Xerox researchers were of the roll-up-one's sleeves and solve the problem mold; when they encountered a problem, they were more likely to dream up a clever engineering solution than to head for Leslie's office for a lesson on the nascent theory. As a result, Xerox pioneered on the practical side of the field, and many of the ideas we will be studying in this text had early forerunners that were developed at Xerox Parc, but never commercialized. By the time the web really picked up steam, the Xerox group had already broken up, although many of those same researchers became leaders in developing today's cloud platforms.

With the benefit of hindsight, one can now see that cloud computing is different from all of these earlier forms of Internet-based computer systems. The cloud reflects a dramatic recent evolution relative to the technologies that predated it, and this evolution has gone in two somewhat opposing directions. On the one hand, the cloud is very elaborate in some ways: servers can send programs over the network for execution within your browser, and there are all sorts of subtle ways for a cloud platform to control the routing of requests and to federate with other cloud provided services, including ones implemented by other companies. And the cloud is massively larger than any prior distributed computing infrastructure: even a few years ago, one rarely saw clustered computing systems with more than 128 or 256 machines; suddenly, we are casually talking about hundreds of thousands and anticipating millions. The Xerox Parc people certainly did not expect this, even though many of the technologies their team created can now be seen as having evolved into key elements of the modern cloud.

Yet the cloud has simultaneously pulled back in some respects relative to distributed computing. As we saw in the introduction, it is not clear that the cloud can safely support applications like air traffic control, or medical computing systems; if it can, it will not do so in the identical way that earlier distributed systems did so. Cloud computing systems, indeed, often guarantee even less than the early networked systems did, and far less than the distributed systems of the 1990s. The evolution to the cloud, in effect, has been a mixture: expansion of things that work, but also a retrenchment from things that did not scale well, even if this entailed a loss of desirable properties.

For example, in the early days of the cloud one tended to view the network as a separate technology from the data centers and clients it linked: the network moved

retical foundations that side of the field. Today, Lamport's theories of distributed computing are widely recognized as foundational steps towards solving problems of consistency, fault-tolerance and coordination. Few people have had more impact on any part of computer science.

the bits; the applications made sense of them. Cloud computing has revamped this thinking in many ways: today, the network is controlled to a growing degree by the cloud applications that depend on it, and those applications actively reprogram the network so as to route traffic, filter unwanted messages, and to host content in ways optimized for what the cloud applications need to do. Thus the network has become a true partner of the cloud, and this has begun to force a rapid evolution of network routing protocols and hardware.

In contrast, the weakening of guarantees and consistency properties that we discussed earlier would be an example of a retrenchment: in these areas, the cloud does less than the massive transactional data centers it emerged from were doing ten years ago. Those platforms had simply expanded as much as they could, and when cloud developers realized that they needed to expand by factors of hundreds or thousands of times more, they decided that if trying to provide strong guarantees was turning out to be hard, they would simply have to weaken the guarantees and find ways to live with the consequences.

2.1.2 Is the Cloud a Distributed Computing Technology?

Reduced to the simplest terms, a distributed computing system is a set of computer programs, executing on one or more computers, and coordinating actions by exchanging *messages*. A *computer network* is a collection of computers interconnected by hardware that directly supports message passing and implements routing protocols, so that the messages have a reasonable likelihood of reaching their destinations.

Most distributed computing systems operate over computer networks, but one can also build a distributed computing system in which information flows between the components by means other than message passing. For example, as the relentless advance of computer architectures shifted from raw speed to multi-core parallelism in recent years, a realization emerged that even a single computer is more and more like a small distributed (clustered) computer system of a few years ago. Future operating systems for multi-core processors may turn out to have more in common with Lamport's style of distributed systems than with the single-core operating systems that of just a few years ago, because when those single-core systems are modified to run on multi-core hardware, they turn out to waste huge amounts of time on synchronization and shared memory management. A distributed system architecture can potentially avoid both problems.

In the introduction we used the term *client-server* to refer to a situation in which one computer requests services from another. The client computing system is just your laptop, desktop, pad computer or even your mobile telephone: whatever device is running the application that you, the user, interacts with. The server, of course, is the application that runs at Amazon or Google or Microsoft or any of a dozen other companies and handles your requests. In fact, as we will see, even a simple request might require cooperation by many servers, and those might not even be running at any single place or controlled by any single company. Moreover, for most kinds of request you can initiate, there are vast numbers of server systems that can field

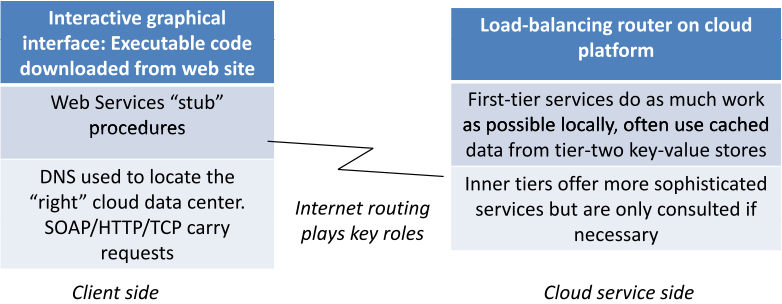


Fig. 2.1 Overall architecture of a cloud computing application

<p>Load-balancing router: Role is to spray requests over available first-tier service instances. Desirable properties include proximity (use the right data center for this user), affinity (if possible, requests from a given client should route to the same server), load balancing, effective use of elasticity.</p>
<p>First-tier services are limited to using soft-state or running without any state at all: on restart, any temporary files or data will be wiped away. They make extensive use of key-value stores and caches running at similar scale in the second tier of the cloud.</p>
<p>Inner tiers offer more sophisticated services but are only consulted if necessary. These often include databases, large precomputed index files, etc. Some inner tier services use strong consistency models, such as the ACID model or snapshot isolation, but these are costly and hence the first-tier shields the inner ones from load.</p>
<p>Infrastructure services manage the ensemble, launching new services or shutting down active ones in response to shifting load patterns and failures. They may do this without warning, especially for services in the first-tier.</p>
<p>Back-end applications run batch-style, often on very large numbers of machines with very large data sets. Using tools like MapReduce or Hadoop, they analyze those data sets and create helper files that will be used later by the first-tier.</p>

Fig. 2.2 Elements found within a cloud computing data center (see also Fig. 1.4)

the request; an important part of cloud computing involves routing each request to a good server that has a chance of responding rapidly and, one hopes, correctly.

Let us walk through a simple example to see how this client-server pattern plays out, and to better understand whether we are looking at networked applications or true distributed computing systems. Suppose that you use your mobile phone or a web browser to query the Google Maps service. While the mobile phone “app” may not look much like a web browser, in fact both of these cases do involve web browsers; they simply have different configurations that determine whether or not the web navigation buttons are displayed. So, in either case, we are looking at an application defined by the combination of the browser on the client’s computer, and the Google Maps server.

Figures 2.1 and 2.2 illustrate the software layers and protocol stacks that are used on the client and data center sides of the application (Fig. 2.1), and the way that the cloud data center breaks the application into a series of *service tiers* (Fig. 2.2). We’ll look at both perspectives in much greater detail in the pages that follow.

These interact through a fairly typical client-server protocol. The browser starts by parsing the URL (<http://maps.google.com/maps>). It extracts the name of the server, in this case, maps.google.com, and then asks the local Domain Naming System (DNS) to map the name to an IP address. Next, the browser makes a connection to this address, using the TCP protocol: the Internet protocol that supports sending streams of bytes reliably, meaning that no data are lost and that data are delivered in exactly the same order as they were sent. TCP has other features too, such as flow control, security, failure detection, but we do not need to worry about those right now.

So, the browser makes a connection to the server. The server sends back a web page back, which it renders. Of course web pages can be quite elaborate, and Google Maps uses some very sophisticated features. Thus these map pages have subpages, parts that can zoom and otherwise interact with the user, pushpins that open up to reveal photos of various sites, etc. Some pages have blanks that the user can fill in: in the case of maps, these let you type in your home address so that you can click to ask for driving directions. What happens here is that the browser creates a little web page that encodes the address you typed in, and then sends it to the server, which sends back a new page: take a left as you leave your driveway, drive 0.3 miles to the entrance of Route 81 North, etc.

Moreover, browsers are getting more and more elaborate; recent innovations allow a server to send a small program to the browser (using packages like Adobe Flex, Silverlight, Java Fx and others). These programs can run animations or interact with the user in very flexible fast ways, which would obviously not be feasible if the server had to compute each new image. Thus the distinction between a web page that the cloud generates, sends to the client, and that gets displayed passively on the client; and a program that runs on the client consuming data from a server has begun to vanish: to a growing degree, the cloud is able to put computation where computing is the sensible thing to do, data where data are most conveniently stored, and to move data to the computation or computation to the data in very flexible, easily implemented ways. Developers focus on functionality and think in terms of speed and responsiveness, and the technologies needed to map their best ideas into working applications are readily available.

This trend towards increasing sophistication is evident in other ways, as well. We are seeing that a client system may be a very elaborate platform, with data of its own (perhaps preloaded, perhaps downloaded and cached, or perhaps captured from cameras, microphones, or other devices), rendering web pages but also capable of running elaborate programs that might be embedded into (or linked to) web pages, and accessing data not just from the main server platform but also from other web sites, such as web sites maintained by content hosting companies or advertising intermediaries. The client could also be mobile, which can have important implications too: a mobile system might experience varying connectivity, and could even see its IP address change over time (for example, if you take your laptop from your office to the coffee shop up the street, the IP address assigned to it would change each time it rebinds to a different access point).

Let us think for a moment about how client mobility impacts our basic Google Maps scenario. Suppose that you are using Google Maps, but doing so as you move

about. Google Maps needs a way to recognize that the same machine is involved in this series of requests. It cannot just use the IP address for that purpose, since that changes. Accordingly, it does the obvious thing: the application leaves a small file on your computer, called a *cookie*. The file can contain pretty much anything, but generally is kept small to minimize transmission costs: when you connect to the Google Maps service, the cookie (if any is available) is included with the initial connection request. But now think about the role of the network: previously, we described the job of the network in simple terms: it connects this IP address to that IP address, trusting the DNS to map host names to addresses, and moves bytes back and forth. Our mobility story is revealing extra complexities: the IP address of the mobile device may be changing as you move about; the cookie uniquely identifies you, but is buried in the application data stream. A seemingly simple networking role suddenly starts to look unexpectedly complex!

To appreciate the implications of these issues, let us briefly drill down on precisely how cloud computing systems and network routing interplay. As we do this, keep in mind that many cloud applications maintain some form of continuous connection to the servers, so that as you move about, they can update the information shown to the end-user. Thus, your mobile device is moving, and yet there is a form of continuity to the connection. We will see that the combination of issues that arises leads to a really complicated picture of network routing, and one that can come as a real surprise to readers who learned how IP networks operate even as recently as a five years ago.

Notice that from the perspective of the network, the IP address of the mobile device could change abruptly as the device moves around: first, it had an IP address assigned by the user's company, then suddenly it became an IP address assigned by Gimme! Coffee. And now the IP address is one assigned by T-Mobile. The application-specific cookie does establish continuity here, yet the network router will not see those cookies; it only sees the IP addresses.

In fact, the end-host addressing issue is even more complex than we are making it sound, because of the prevalence of what are called *network address translation* or NAT technologies. Many devices, such as wireless routers, have a single outward-facing IP address but support a small interior network, and when this occurs, they often use their own IP addressing space for the interior machines. What this means is that if I sit down in Gimme! Coffee in Ithaca, and then check my IP address, there is a good chance it will be 192.68.1.x, where x is some small number like 2 or 3. Yet the Google system sees a very different IP address, perhaps 176.31.54.144: that assigned to Gimme! Coffee by its ISP. The NAT box translates back and forth: when a packet is sent from an interior machine to the outside, the box replaces the IP address and port number in the packet with the IP address of the NAT box and a dynamically selected port number that it associates with the original sender. Later, when a reply packet is received, the NAT box maps in the other direction. NAT functionality is very common today, and runs so quickly that we are completely unaware of it. Indeed, we would have run out of IPv4 addresses long ago if we did not have NAT capabilities. With NAT, each router can be a gateway to a huge range of IP addresses—perhaps, a whole cloud computing data center full of them.

In newspaper articles about the Internet one reads that an IP address is like a street name and house address, but here we can see that a single IP addresses might be in use by a great many computers. For example, as just noted, the local Gimme! Coffee here in Ithaca assigns IP addresses that look like 192.68.1.x. Well, it happens that the computer on which this chapter was being written *also* thinks its IP address is 192.68.1.2, even though I am nowhere near Gimme! Coffee right this minute. The reason is that 192.168.1.2 is perhaps the world's most heavily reused address: most wireless routers are preset to assign addresses starting with 192.68.1.1, which the router uses for itself. My router works this way, and it assigned my laptop computer the next address in the sequence. All over the world there must be millions of wireless routers, each of which uses 192.68.1.1 for itself and each of which has assigned 192.68.1.2 to some local user's laptop. The situation is similar to a world in which every town has a Main Street, and every Main Street has a building at number 2.

With NAT-enabled routers on the path, the Internet route between two points is really an implicit part of the address. If we trace a packet from your machine to Google Maps, we will see that your machine first routed it to the wireless router at Gimme!, which replaced the source address with its own address. Next it forwarded that packet through the Gimme! cable modem, which may even have done a second remapping very similar to the first one. Now your packet is inside the local Internet Service Provider (ISP), which maintains routing tables telling it that to get from here to Google.com, packets should be passed to the AT&T network, and so forth. Step by step, the packet advances (just as you were taught to expect a few years ago), but the source and destination addresses and port numbers potentially change each time the packet passes through an active routing element. The route from Gimme! in Ithaca to the nearest Google Maps data center probably involves 15 to 20 hops, and as many as five or six of these could remap the addresses as part of their routing functionality.

The sense in which the route is “part” of your computer's address is that for Google's servers to respond to your request, packets need to be sent back using the sender address in the IP header. This, of course, will have been modified by each NAT box in the route. Thus, the Google response travels to the last NAT box that your packet traversed on its way into Google, then from that box to the next one closer to you, and so forth until we get back to the router at Gimme!, which replaces the address with the internal IP address for your machine on the Gimme! wireless network. If you were to move your machine to a setting with a different wireless router, you might actually be assigned the same IP address (192.61.1.x) and yet packets sent by Google to your previous address will not get to you anymore. RSS feeds and Internet radio feeds and movie streams and podcasts will all fail, and need to be restarted from the new location.

So we have traced the route from Gimme! to Google and back. But two more questions arise: which Google data center was used? And which machine? The former question matters because Google.com is a single name by which all of Google's infrastructure is named, yet Google operates hundreds of data centers, worldwide. Obviously, the company would typically map a particular user's request to the nearest data center, which it does by forcing the DNS to hand out different IP addresses

to different users, depending on where they originate—something it learns by looking at their IP addresses! Given the ubiquity of NAT boxes, one suddenly appreciates just how complex that determination can be: a million distinct machines, all over the globe, may claim that their IP address is 192.68.1.2, and that they need to send a packet to Google.com—meaning, the nearest Google data center.

As it happens, this localization problem is solvable, but not in a trivial way. Any textbook needs to limit its scope and this one will not cover networking technologies in great detail. In a nutshell, researchers have found ways to correlate delay to path length, and cloud computing data centers take advantage of this to create a kind of network coordinate system a bit like a GPS coordinate; two widely cited examples of systems offering this functionality are Vivaldi (Dabek et al. 2004) and Meridian (Wong et al. 2005). The basic idea is to establish some set of *landmarks*. The client system measures its round-trip times and perhaps some aspects of the route used to estimate its distance from each landmark, then encodes these data into a tuple that functions much like a 3-dimensional map coordinate in the real world. Given two such tuples, one can estimate the network latency between the machines at each location. And this functionality can also be embedded into the DNS name resolution mechanism: based on observations of timing, a company like Google can localize your DNS to at least some degree, and then use that localization to instruct that particular DNS server to give out this particular IP address as the mapping of Google.com. Thus Google is able to direct your machine to the closest data center, with one or two other options offered as good backup possibilities (DNS mappings typically resolve to a primary IP address but list some number of backup options as well). Notice that we’ve arrived at a definition of “closest” that minimizes network latency and maximizes bandwidth; the best choice of data center might not be the one that is physically nearest to your location.

So you launch your mapping application. It needs to map Google Maps to an IP address, and asks the DNS for help. The DNS does a quick request to Google, asking for the IP address to which your requests should be directed. From then on, at least until this DNS record expires, the DNS will give the same answer repeatedly. This, then, gives Google a way to direct queries from Chinese users to its Hong Kong data center, queries from India to a data center in Hyderabad, and queries from Ithaca to a data center somewhere in New Hampshire, or Canada: Google likes to place data centers in settings that are remote, where power is cheap, where the weather is as cool as possible, and with good networking capabilities.

Google can play with the expiration values on these DNS records: to exert very fine-grained control over the DNS mapping, it hands out DNS records that expire instantly; in such cases, every single host-to-IP-address mapping for machines at Google.com will need to be forwarded to Google. That can be slow, but gives Google total control. If Google’s servers believe that it is safe to do so, on the other hand, they can hand out DNS records with much longer expiration times. Performance for DNS mapping will be much improved, but Google cannot retract those mappings other than by waiting for them to expire.

What happens after a packet reaches Google in New Hampshire? Here, a further complication arises: Google has its own specialized routers to handle incoming traffic; they look at each packet and select a Google server to handle it on the basis of

a wide range of criteria: the requested service, the current configuration of the data center, load estimates, which server you last talked to, etc. A single data center can also host many .com names: [YouTube.com](https://www.youtube.com) and [Google.com](https://www.google.com) could route to the same data center. In some sense, we should understand the IP addresses and machines as being a virtual overlay, superimposed on the physical network and physical data center hardware, but accurately *emulating* a collection of networks and a collection of distinct, dedicated computers. This facilitates creation of new cloud computing applications. In one of the more common ways of creating a service, the developer implements the service on a development system but then uploads a virtual machine image, and the cloud deploys as few or as many copies as it likes, spraying incoming requests over the replicas to spread load.

Google, then, as a company hosting large numbers of virtualized services (including these kinds of virtualized web site), has a great deal of control over routing, and it needs that control in order to ensure that you as the end-user will have an acceptable experience and that it, as the provider, will make the best possible use of its data center resources. Part of this is determined by IP addresses, but because those are virtual, other factors also come into play: the actual route that was used from a client into the DNS hierarchy, the route from the DNS server that translated the address to Google, and even the cookies that identify the actual client. Ideally, Google needs to use all of these elements to control routing so as to optimize such aspects as performance, cost of the solution and security against disruption by malfunctioning applications. Moreover, Google will make an effort to route the stream of requests originating in your mobile device to some single server that will handle them all, unless a failure or some other kind of management event happens to force a reconfiguration. This way, if that server maintains state on your behalf, you get a continuous evolving story from your mobile application. And yet because the network does the routing, the only parts of this story that the IP network itself observes are the source and destination IP addresses and port numbers in use on each leg of the current path.

Here we encounter an example of a current cloud computing challenge. Today, Google is not able to inspect the contents of incoming packets until they actually reach one of its data centers. Thus, suppose your mobile application loses its connection and tries to reconnect. Perhaps its IP address has changed, and very likely it will find itself talking to a different DNS server than it was using moments earlier. That server may not have any knowledge of which Google data center you were talking to. Thus, your reconnect might show up at a very different Google data center than your previous packets were talking to.

When the reconnect arrives, the load-balancing component of the data center can inspect the packet, find the cookie, and perhaps make an association to the prior activity, in which case the end-user experience will be nearly seamless. But it may also be too late to preserve application continuity, and that seems especially likely if because of the changing IP address used by your mobile, the reconnection request was routed to a different data center. In that case, you'll experience a very annoying loss of continuity. If your friend were to tell you about some other mapping system that never breaks down this way, you might switch.

Let us compare the way this works today with the ways it might be improved, as an opportunity to peer into the way that cloud computing makes tradeoffs right now. To be concrete, assume that the client was watching a movie and that it was streaming from a server *S* in Google's YouTube system, and that just as we reached time 10:27 in the movie, the connection broke. What are the options for masking this problem?

Notice that there can be many reasons for a connection failure: the client may have lost connectivity to the Internet or been shut off, the server may have failed or been "elastically" shut down or migrated (really, the same thing, since the cloud migrates a server by killing one instance and starting a different one), the Internet routing system may be having issues, etc. In our toy scenario, the client is still up. So it issues a completely new connection request to YouTube, but now the URL encodes the offset into the stream. Ideally, the player resumes seamlessly; now some server *S'* is probably handling the request, and playing back starting at time 10:28.

How long will this take? To mask the outage the new connection would need to be up and running within a few seconds. But of course *S'* may not have a cached copy of the movie, in which case it would need to fetch a copy from the global file system. *S'* might also need to recheck the client's credentials, something *S* would have done during the initial connection. Thus the client probably will see at least some delay during this process: very likely long enough to exceed the few seconds of tolerance. Thus it is not uncommon to need to restart that sort of transfer manually today.

One could do better. For example, if the cookie the client presents to *S'* has some sort of "recently validated" token in it, *S'* could cut that step short. But this still would not eliminate delays associated with the film not being locally available.

Suppose that *S* is still running, and we lost the connection entirely because of mobility: one of those changing IP address scenarios. In this case, if the protocol used between the cloud and the video player includes some kind of ticket representing the original server, *S*, the player could reconnect to *S* under some sort of server name: S.YouTube.com. Your local DNS would not know how to resolve this name, so it would be passed to YouTube (that is, to Google.com as the host for YouTube.com), where an attempt could be made to reconnect you to your original server, *S*. Thus with a bit of DNS trickery, the client ends up talking to the original server, which has a warm cache, and we get playback more or less instantly. Notice that "*S*" doesn't have to be a name bound to a specific physical server, either: in modern computing settings, a computer can have more than one name, and those names can be moved around. So "S.YouTube.com" is really more of a logical name for a virtual server, not necessarily the physical name of a specific machine in a specific location.

In fact one can even get this behavior when a server migrates. Suppose that *S* failed but that our cloud system assigned its roles to *S'* prior to *S* shutting down, so that *S'* has a chance to prefetch the movie and copy the credentials. In such cases, one can actually splice a new TCP connection to an old one seamlessly, so that the connection never breaks at all. The same techniques can also support IP address migration on the client side (we will discuss this in more detail in Chap. 4). These schemes can be surprisingly lightweight (the former approach is transparent but a

bit more expensive, the latter requires some help from the endpoint application (the video player in the client, or the server), but imposes almost no additional cost at all, and can reestablish connections in tens of milliseconds. Thus one could actually imagine systems that completely mask disconnection and offer continuous streaming availability so long as the outage was one of these recoverable cases.

We do not see such mechanisms in use today, for purely pragmatic reasons. First, as noted, even if we did use these techniques, there would still be cases they cannot handle: for example, if a client vanishes, how long should its server wait before giving up on the connection? Thus the disruption scenario would need to be pretty common relative to these unrecoverable ones in order for the schemes we have cited to be beneficial: if they could overcome 99% of all crashes, they would make obvious sense; if they only cover 6% of them, far less so. One can guess that the benefit must not be all that high since today's cloud platforms are quick to embrace improvements that end-users would notice.

Thus, rather than use fancy failover schemes, today we see other sorts of reconnection hacks. For example, right now video playback is the dominant case where such issues arise, and realistically, video playback works reasonably well without needing these fancier reconnection solutions. Applications that need better behavior, such as voice-over-IP telephone connections, avoid relaying data through the cloud: Skype, for example, uses a cloud platform to make the initial phone connection, but after that tries to use direct point-to-point connectivity between the callers. Internet conferencing solutions are just not all that popular. But if use of such technologies grows, the cloud could easily evolve to mask disruptions more effectively.

Our example illustrates a pattern we will see throughout this textbook. Today's cloud is a world of tradeoffs that optimize certain kinds of applications; tomorrow's cloud is likely to make these same tradeoffs in different ways, depending on the way that demand in the marketplace evolves, and on the kinds of application that are bringing in the lion's share of the revenue. One reason for learning about techniques like recoverable TCP connections is that even if they are not widely used today, they could be the key to success in some important setting tomorrow. Moreover, while it made sense in the past to talk about distributed systems as distinct from networked ones, the cloud is so complex that these simple distinctions no longer can be applied. Recalling Lamport's point, in the cloud we all "depend" on a great many components. If any fails, our computers could become unusable.

Figure 2.3 shows the overall architecture of a typical cloud computing application, highlighting the various routing and address translation layers we have discussed. Sophisticated applications control the DNS mapping from host name to IP address, typically seeking to direct each user to the nearest data center, and then as the request reaches that data center, redirecting it again to reach a lightly loaded machine in the first tier.

Figure 2.4 shows some elements one might find inside a typical cloud computing data center. Here we see the router and DNS at the top left, vectoring requests to the highly scaled and elastic row of first-tier services. The elements of this layer can be spun up or shut down very abruptly as demand varies. A second row of stateful services runs behind the first-tier applications providing somewhat greater functionality, but at higher cost and with less ability to scale up or down as quickly. These

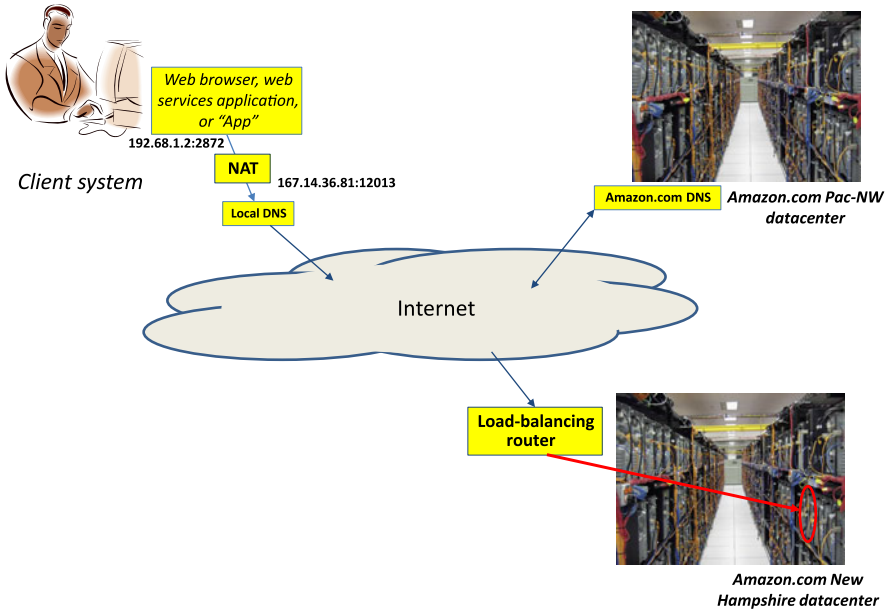


Fig. 2.3 Overall architecture of a cloud computing system focusing on the routing of a user's request to a data center. The user's initial request to [Amazon.com](https://www.amazon.com) is vectored by the local Domain Name Service (DNS) to a DNS service managed by Amazon, which selects a data center to which the user's traffic should be routed. In this illustration, the DNS that receives the mapping request happens to be the one running at one of Amazon's Pacific NorthWest data centers, but the user will be redirected to an Amazon data center in New Hampshire, where a load-balancer will in turn redirect it to a specific server. There may be one or more network address translation devices on the path; these convert between an internal IP address and port number and an external IP address and port number, with the effect of greatly enlarging the address space

in turn are supported by various databases and files. In the back-end of the system, a variety of offline applications collaborate to maintain those databases and files, manage the infrastructure, share data over large numbers of nodes, etc. The most dramatic level of scaling occurs in the first tier, which absorbs as much of the work as possible, seeking to offer locally responsive behavior even if this may entail running on stale data or not waiting for a slow component to respond, which are just two of the many cases in which a cloud system might behave in an inconsistent and perhaps insecure manner.

Notice that any particular data center will have its own set of services and that many cloud systems would look quite different from the one shown in the figure. Cloud systems share an overall style, but the details depend very much on the particular applications being supported and the platform developer's design choices. Indeed, most cloud systems evolved independently to solve specific problems: the [Amazon.com](https://www.amazon.com) system to support its e-commerce web sites, the Facebook system to support its social networking applications, Google's to support search. Then as time passed, they grew more and more general and began to host third-party content.

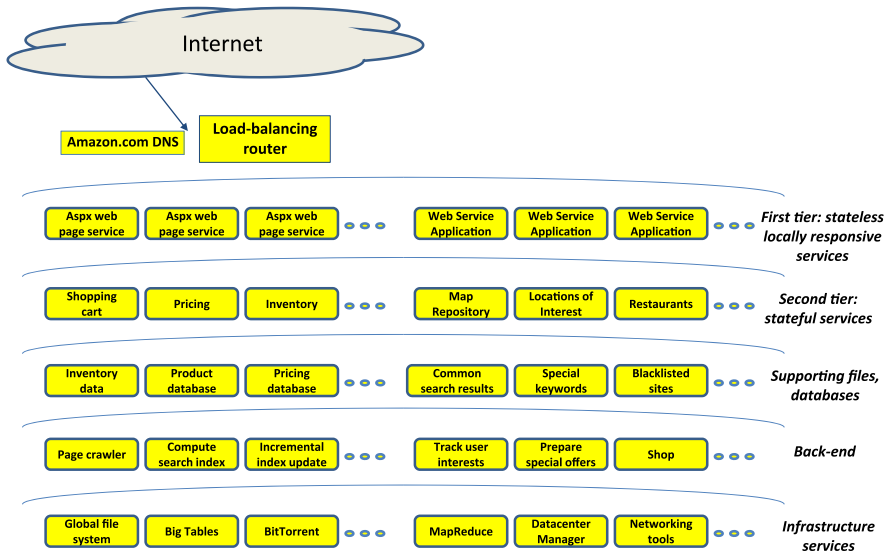


Fig. 2.4 Interior of a cloud computing system, showing some of the major functional tiers on which the textbook will focus

Today, there are a few dozen major cloud computing platforms, each with its own specialties, and while there is a great deal of functional overlap between the resulting systems, there are also big differences.

In this text we will not focus on any specific cloud platform but will try to abstract major questions and look at the fundamental issues that are posed. On the other hand, we will look closely at some of the most widely discussed cloud technologies, such as cloud computing file systems (GFS for Google, Zookeeper from Yahoo!, as well as others), key-value stores (BigTable in the case of Google, Dynamo for Amazon), locking services (Chubby, which uses the Paxos protocols), compute engines (MapReduce, also widely used through the Hadoop open-source version), data distribution tools (BitTorrent), and others. Our hope is to strike a balance: to expose the reader to some of the “big name” technologies that are best known and most widely deployed, while also pushing to deeper questions such as just how one offers consistency in a data replication service, or a locking service.

2.1.3 What Does Reliability Mean in the Cloud?

We have been fairly informal about our terminology. let us pin things down. When computer systems exchange messages over a network, we say that a *protocol* is running between the applications that participate. More formally, we will use the term “protocol” in reference to an algorithm governing the exchange of messages, by which a collection of processes coordinate their actions and communicate information among themselves. Much as a *program* is a set of instructions, and a *process*

denotes the execution of those instructions, a *protocol* is a set of instructions governing the communication in a distributed program, and a distributed computing system is the result of executing some collection of such protocols to coordinate the actions of a collection of processes in a network.

We keep using the term *reliability* but this needs a bit more precision too. After all, reliability can have many meanings. Here are a few that matter here:

- *Fault tolerance*: The ability of a distributed computing system to recover from component failures without performing incorrect actions. (Of course, the application designer gets to define correct and incorrect behavior.)
- *High availability*: In the context of a fault-tolerant distributed computing system, the ability of the system to restore correct operation, permitting it to resume providing services during periods when some components have failed. A highly available system may provide reduced service for short periods of time while reconfiguring itself.
- *Continuous availability*: A highly available system with a very small recovery time, capable of providing uninterrupted service to its users. The reliability properties of a continuously available system are unaffected or only minimally affected by failures.
- *Recoverability*: Also in the context of a fault-tolerant distributed computing system, the ability of failed components to restart themselves and rejoin the system, after the cause of failure has been repaired.
- *Consistency*: The ability of the system to coordinate related actions by multiple components, often in the presence of concurrency and failures. Consistency underlies the ability of a distributed system to emulate a non-distributed system. Later, though, we will see that there are many ways to implement this kind of emulation—many ways to implement consistency guarantees. Thus when we commented that consistency is an issue in the cloud, and talked about CAP, that really relates to one particular consistency model (the one used in database settings that support ACID guarantees). One can accept that this form of consistency will not work in the cloud, and yet still build cloud-scale solutions that have strong consistency properties.
- *Scalability*: The ability of a system to continue to operate correctly even as some aspect is scaled to a larger size. For example, we might increase the size of the network on which the system is running—doing so increases the frequency of such events as network outages and could degrade a “non-scalable” system. We might increase numbers of users, or numbers of servers, or load on the system. Scalability thus has many dimensions; a *scalable system* would normally specify the dimensions in which it achieves scalability and the degree of scaling it can sustain.
- *Security*: The ability of the system to protect data, services, and resources against misuse by unauthorized users.
- *Privacy*: The ability of the system to protect the identity and locations of its users, or the contents of sensitive data, from unauthorized disclosure.
- *Correct specification*: The assurance that the system solves the intended problem.
- *Correct implementation*: The assurance that the system correctly implements its specification.

- *Predictable performance*: The guarantee that a distributed system achieves desired levels of performance—for example, data throughput from source to destination, latencies measured for critical paths, requests processed per second, and so forth.
- *Timeliness*: In systems subject to real-time constraints, the assurance that actions are taken within the specified time bounds, or are performed with a desired degree of temporal synchronization between the components.

Underlying many of these issues are questions of tolerating failures. Failure, too, can have many meanings:

- *Halting failures*: In this model, a process or computer either works correctly, or simply stops executing and crashes without taking incorrect actions, as a result of failure. As the model is normally specified, there is no way to detect that the process has halted except by timeout: It stops sending “keep alive” messages or responding to “pinging” messages and hence other processes can deduce that it has failed.
- *Fail-stop failures*: These are accurately detectable halting failures. In this model, processes fail by halting. However, other processes that may be interacting with the faulty process also have a completely accurate way to detect such failures—for example, a fail-stop environment might be one in which timeouts can be used to monitor the status of processes, and *no timeout occurs unless the process being monitored has actually crashed*. Obviously, such a model may be unrealistically optimistic, representing an idealized world in which the handling of failures is reduced to a pure problem of how the system should react when a failure is sensed. If we solve problems with this model, we then need to ask how to relate the solutions to the real world.
- *Send-omission failures*: These are failures to send a message that, according to the logic of the distributed computing systems, should have been sent. Send-omission failures are commonly caused by a lack of buffering space in the operating system or network interface, which can cause a message to be discarded after the application program has sent it but before it leaves the sender’s machine. Perhaps surprisingly, few operating systems report such events to the application.
- *Receive-omission failures*: These are similar to send-omission failures, but they occur when a message is lost near the destination process, often because of a lack of memory in which to buffer it or because evidence of data corruption has been discovered.
- *Network failures*: These occur when the network loses messages sent between certain pairs of processes.
- *Network partitioning failures*: These are a more severe form of network failure, in which the network fragments into disconnected sub-networks, within which messages can be transmitted, but between which messages are lost. When a failure of this sort is repaired, one talks about *merging* the network partitions. Network partitioning failures are a common problem in modern distributed systems; hence, we will discuss them in detail in Part III of this book.
- *Timing failures*: These occur when a temporal property of the system is violated—for example, when a clock on a computer exhibits a value that is unacceptably far

from the values of other clocks, or when an action is taken too soon or too late, or when a message is delayed by longer than the maximum tolerable delay for a network connection.

- *Byzantine failures*: This is a term that captures a wide variety of other faulty behaviors, including data corruption, programs that fail to follow the correct protocol, and even malicious or adversarial behaviors by programs that actively seek to force a system to violate its reliability properties.

Readers might want to pause at this point and consider some cloud computing application that they often use: perhaps, our map-based applications, or perhaps some other application such as Shazam (a mobile application for recognizing music from small samples), or Twitter. Look at these notions of reliability, and imagine yourself in the role of a lead developer creating that application. Which forms of reliability would matter most, if you want to ensure that clients can count upon your solution as a reliable mobile tool that would play a big role in their mobile lives? What obstacles to that form of reliability can you identify, and where would be your best hope of addressing the resulting requirements? The client system? The Internet itself? Or the cloud computing system? You'll see that even without knowing how cloud computing systems really work, you actually can reason about a question like this. Moreover, you can probably convince yourself that the answer necessarily involves many moving parts: for any sophisticated cloud functionality, the client can (and probably must) provide some of the needed functionality, the network others, and the cloud system itself has its own role to play. The cloud solution may be the conductor of the orchestra, but without ways to control each of these components, it would not be possible to build reliable cloud applications.

2.2 Components of a Reliable Distributed Computing System

So, at the end of the day, where does reliability really come from? There isn't any single magic answer. Reliable distributed computing systems are assembled from basic building blocks; in general, those building blocks can experience failures, and depending on the nature of the problem and its severity, the system may be able to overcome the issue, or it may shut down or malfunction. Thus, the Google Maps service can tolerate the crash of a cloud server running the map application: if this happens, the client requests will be reissued and routed to a different server. Of course, some information might be lost, but the map application is designed so that such events have minimal disruptive impact. But suppose that a recent accident has closed a bridge on the route you are taking. Will your Google Maps unit know? This is less certain: while mapping systems track such events, to get this information to you the server you are talking to will need to see the update promptly; your mobile device will need to talk to that server; the network will need to be stable enough to let it download the relevant data. Over time, the cloud can certainly do all of these things. If split-second decisions are involved, however, we run into a situation for which the cloud as currently designed just is not ideal (obviously, some systems,

like Twitter, are specialized in replicating some forms of information, but those are specialized and not typical of the platforms in the widest use). The point here is that even if a map application is perfectly reliable, you might perceive it as unreliable if it fails to warn you of a closed bridge. Reliability must be judged by the end-user experience, not in terms of any kind of narrower, more technical definition focused purely on the reliability of individual subsystems.

Staleness of underlying data is just one of many issues of this kind. Cloud systems are also at risk of being confused by data that were incorrect in the first place, or that became corrupted once they entered the system. Moreover, systems like the ones we have described sometimes turn into targets for hackers or other kinds of intruder, such as foreign intelligence organizations or corporate espionage teams. Attacks can come from the outside or the inside: not every single employee at the company necessarily buys into the “do not be evil” motto. One can only protect against some of these kinds of issue, and even when we can, the mechanisms are sometimes expensive or use techniques that do not scale very well. Some forms of reliability, in effect, go beyond what a cloud platform is designed to do.

Your job, as the designer of a reliable cloud computing application, will be to start by thinking hard about what reliability needs to mean for the end-user, taking a holistic approach: you need to think about what the end-user is trying to accomplish, and to view your system as a black box, thinking about its properties without rushing to make decisions about how it will be built. You’ll need to set reasonable goals: a system might be able to protect itself against some forms of problems, but perhaps not others; some properties may even be mutually exclusive. Moreover, you need to strike a balance between the costs of the reliability properties you desire to offer and other properties, such as the speed and scalability of your solution. Even the complexity of the solution should be viewed as an issue: a complex reliability-enhancing technique could actually decrease reliability by being hard to implement. A simpler solution that offers less coverage may still be more reliable if it can be completed on schedule and on budget, tested more carefully, and if the limitations do not pose frequent and grave problems for its users.

Our job in this text will not be to address this software engineering task. Instead, we will take up the challenge at the next step: given sensible reliability goals, we will focus here on the technical options for achieving them. In effect, we will create a menu of options, within which the savvy designer can later pick and chose. Reliability will not turn out to be a one-size-fits-all story; for any given purpose, it demands choices and often requires compromises. Indeed, sometimes reliability comes as much from steps taken outside of a system as it does from the ones we take within it. For example, when designing health-care systems and air traffic control systems, one explicitly recognizes that even the most reliable system will sometimes fail. Accordingly, we use a fail-safe mindset: we design the solution with safeguards that will ensure the safety of the end-user no matter what may happen within the system. Thus, if a cardiac monitoring system goes offline, one wants it to somehow signal that the technology has failed; this is far preferable than trying to engineer a system that can never fail.

2.3 Summary: Reliability in the Cloud

Our simple example leads to several kinds of insight. First, we have seen that cloud computing systems need to be understood in terms of three distinct components, each of which has a distinct role (and each, of course, is itself structured as a set of components). The first component is the client application: perhaps a browser, or perhaps some sort of program that uses the same protocols employed by browsers, issuing requests to cloud-hosted services and receiving replies from them. These requests and replies take the form of web pages, but in a very simple format intended for machine consumption, not humans. The second component is the network itself, which moves the data, and is built as a complex assemblage of routers, together with a small set of network-hosted technologies like DNS (to map host names to IP addresses) and BGP (one of the protocols used by routers to maintain routing tables). Indeed, while we often think of the Internet as a kind of fancy wire carrying data from our laptops to the data centers that provide services like email, the better mental image would be more akin to those very fancy Swiss watches stuffed with springs and gears and all sorts of strange looking twisty things, all moving in a completely implausible ballet. But this Swiss watch is vastly larger and has far more moving parts! Finally, the third major component of a cloud computing system is the cloud computing data center itself. That data center reaches out to control the routing from your computer to its point of ingress, and then the ingress router examines your packets and redirects them to some server. That server, talking to other servers, builds a response for you: you're a client of the server, but it is a client of other servers and services.

Next, we have seen that the system per-se is really a part of a bigger story. The end-user is concerned with that bigger story, not the minutia of how the system works internally. Reliability is an end-user property; the end-user's goals and perception should shape the technical goals and decisions we make as we look more closely at each of the three components of the solution.

When we ask about reliability, we have seen that we are being sloppy: many people have an intuitive sense that reliability is about availability and quick response, but we might have any of a number of other properties in mind; moreover, some settings actually require some mixture of properties for safety (for example, a controller for an insulin pump should either operate properly, or it should go offline and a warning should sound), while others intend the term in a looser sense. Speaking broadly, we have seen that reliability in the cloud as it exists today is of this latter kind: it can be hard to pin down a precise meaning for the term, and the kind of reliability we are offered comes from many separate mechanisms, each working to overcome certain kinds of issue. These can add up to a convincing illusion of reliability, but can also break down in visible ways, without anything in the cloud realizing that anything has "gone wrong." After all, what does "wrong" mean for a system that does not really define "right?"

To the extent that a reliability property in question is pinned down, there may be many ways to obtain the desired behavior. Among these, we might change the application to require a different property (this can be a good idea if the original

property is unreasonably costly or even infeasible), we might manage to show that a standard cloud service can achieve the desired property, or we might find a non-standard way to implement what we seek, hosting the resulting system on a cloud platform (and taking steps to confirm that the cloud will not do anything that might prevent our implementation for working correctly).

Depending on the situation, not all of these options will be applicable. For example, modifying an application is not always feasible; there are often serious requirements that we just cannot hack around. Building an entire home-brew infrastructure is not a minor undertaking: you will have a big job, and your solution, however good it may be at guaranteeing whatever you set out to promise, may seem to be lacking all sorts of basic features that users have come to expect. Moreover, the experience in the field makes it clear that any approach that deviates too far from the mainstream will later seem hard to maintain, expensive and ultimately will fail. When feasible, the best choice is to just find some way to run the needed solution on a standard cloud infrastructure, but to take steps to confirm that it does what we really need from it—to prove, in effect, that the solution will really solve the problem.

This approach does raise questions, however. First, building a cloud-hosted application that pushes beyond the limits of the standard cloud is not a minor undertaking. As we have observed, and will see in upcoming chapters, the cloud has bought heavily into the end-to-end principle and the CAP conjecture, leaving reliability and security and other similar properties to the client and the server, and leaving the Internet to play one role: moving packets as rapidly as possible. But we will see that sometimes, one can strengthen these kinds of behavior in ways that respect the standards, and yet give us more than the standards can provide. Second, even if our solution works on day zero, we need to worry that as the cloud evolves, something that was implicitly assumed in the application might cease to hold.

A second question really relates to a stylistic choice for this textbook. Would it make more sense to dive deep on specific questions, such as the network routing control problem we discussed above? Or should we focus on design patterns: paradigms that one can study in a more abstracted way, and then later translate back to practice when working on specific applications. On this the author of this text feels conflicted: having built cloud computing services of his own, it is absolutely clear that a book of cloud computing recipes could be hugely successful. Right now, the best way to find out how to do many things is just to issue enough web search queries to stumble on an example in which someone else did the thing you are trying to do, and then copy the code. This is definitely not the best way to learn, and seems certain to propagate mistakes from the random web pages you stumble upon into your own software. Yet this is how one does it today.

Nonetheless, this will not be a book of recipes. Instead, we will opt for the second approach, teasing out abstracted questions that we can focus upon in a clean way, solve, and then either implement as needed, or perhaps package within the Isis² library (Appendix B), if the technique is a bit tricky. In the chapters that follow, we will often pose important questions that admit many kinds of answer. Sometimes we will lay the options out one by one; in other cases we will just focus on the best known answer, or the approach actually used in cloud computing standards and

platforms. As a reader of this text, you will need to think these questions out on your own: is this the right way to pose my application goals? Can my goals be realized in a low-cost way? Given multiple possible realizations, which is the best choice? There will not be any simple formula to guide you to the answers, but an approach that focuses on scalability and simplicity will rarely steer you wrong.

Einstein is said to have remarked that one should strive to make things as simple as possible... but not simpler. Cloud computing systems are unavoidably complex, because they have so many components, connected in such elaborate and dynamically evolving ways. We could lose ourselves within this complexity and not learn very much at all. Within the cloud, we work with a vast array of mechanisms and normally, those mechanisms are working and doing a wonderful job; this is why cloud computing has taken off. Yet our overarching theme in this text is reliability, and the real challenges arise not when all works as hoped, but when something goes wrong: how, then, will the application behave? Today's cloud often opts for the simplest solution that will not trigger a crash and a blue screen. But if we need to make a cloud application more reliable than what you get in the standard way, we should expect that in some cases, there will not be any easy way to avoid grappling with fairly complex issues. Oversimplifying would simply be a recipe for unreliability.

2.4 Related Reading

The slide set used by Eric Brewer for his 2000 PODC keynote talk on cloud computing scalability (Brewer 2000), and the slide sets and remarks of speakers at the 2008 ACM Workshop on Large Scale Distributed Systems (LADIS) are well worth looking at. LADIS, in particular, featured a number of cloud computing experts (see <http://www.cs.cornell.edu/projects/ladis2008>).

Stanford law professor Lawrence Lessig has written several fascinating books on the growth of the Internet and the impact of technology on society (see Lessig 1999).

On the topic of fail-safe application design (see Leveson 1995).

General discussion of network architectures and the OSI hierarchy (see Architecture Projects Management Limited 1989, 1991a, 1991b; Comer 1991; Comer and Stevens 1993; Coulouris et al. 1994; Cristian and Delancy 1990; Tanenbaum 1988; XTP Forum).

Pros and cons of layered architectures (see Abbott and Peterson 1993; Braun and Diot 1995; Clark and Tennenhouse 1987, 1990; Karamcheti and Chien 1994; Kay and Pasquale 1993; Ousterhout 1990; van Renesse et al. 1988, 1989).

Reliable stream communication (see Comer 1991; Comer and Stevens 1991, 1993; Coulouris et al. 1994; Jacobson 1988; Ritchie 1984; Tanenbaum 1988).

Failure models and classifications (see Chandra and Toueg 1991; Chandra et al. 1992, Cristian 1991b; Cristian and Delancy 1990; Fisher et al. 1985b; Gray and Reuter 1993; Lamport 1978a, 1978b, 1984; Marzullo 1990; Sabel and Marzullo 1994; Skeen 1982a; Srikanth and Toueg 1987).



<http://www.springer.com/978-1-4471-2415-3>

Guide to Reliable Distributed Systems
Building High-Assurance Applications and Cloud-Hosted
Services

Birman, K.P.

2012, XXII, 730 p., Hardcover

ISBN: 978-1-4471-2415-3