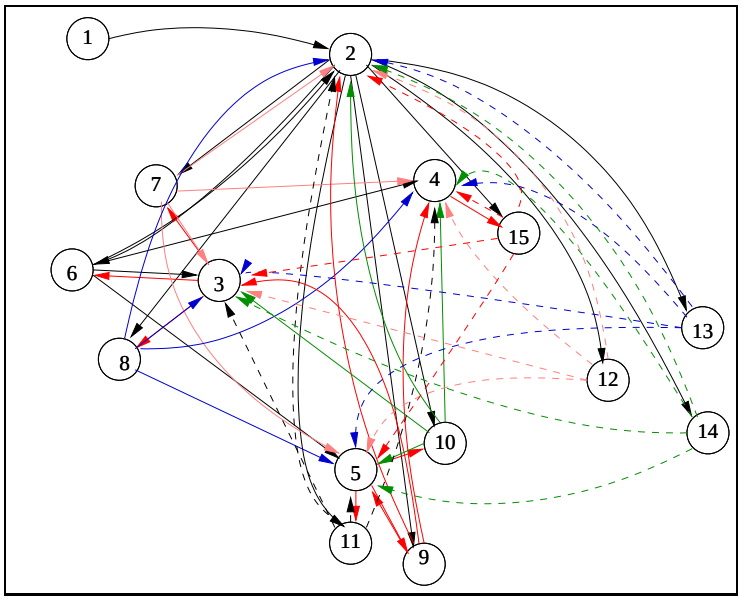
PROMISE YOU A ROSE GARDEN

An essay about system management

Mark Burgess

1 January 2007



A fable

Once upon a time there was an arrow. This arrow was reputed to have been tooled in the armoury of Cupid himself. It issued from a bow, from whose string sprung only pure intentions, and it carried with it a thread of silk so that it could be reeled in again at such a time that is suited the archer. But although the arrow was strong and straight and pure, the arrow's intended target was clad in a special kind of autonomous armour that resisted the thrust of its point, and so, when fired, the arrow was not received, but merely landed close to its target.

Someone close to the target, knowing the nature and circumstances of the armour, saw the arrow shining in the sun, for arrows from Cupid are made of the finest golden material, pure and universal in their appeal. The competitor was so impressed by the arrow, and appreciative of its value, that it fired a sucker in the direction from which the arrow was aimed, attached to it a message saying "if you fire such an arrow at me, I shall not only open my armour to receive it, but return an arrow of my own, made from the best I have to offer."

The arrows were fired, not in violence or coercion, but as an offering of riches. Each had its strings attached and hence the two were bound together. And so they lived, each independent of mind and free of choice, but willingly bound in union of mutual value, all in appreciation of the richness of one another's offers. And they said to one another: "Let us call this state of mutually beneficial union commerce*."*

1 Promises, promises...

A promise is like an arrow, something fired from a human, a business, a computer or other party, to another. Each one carries a claim about its originator, to the receiver (who, by the way, has no direct influence on the content of the promise)[1]. Each promise is aimed at a single recipient, but the promiser can fire similar arrows at any number of targets to repeat the promise. Promises are non-transferable and constrain only the promiser, never the promisee. Duplicating a promise to the same recipient has no effect, but repeating a promise

with a change added would be a promise broken.

These simple ideas are the basic facts about promises. From these we can describe the world in which system administration takes place. Promises are clearly an important part of human interaction, but why talk about promises? Why not the concrete events which realize and fulfill promises? Surely (in our pragmatic materialism) they are all that is of interest?

No, that is a mistake. It is a mistake that derives in part of the great Monitoring Myth that system management has perpetuated for many years, namely that "system management" is all about "monitoring and event detection". Why is that fable a distraction? One reason, of course, is that by the time you observe an event, it is too late to do anything about it. Moreover, there is a huge difference between what you promise and what you need to do to keep a promise during different events and happenings.

> *Promises can be constant, but the recipe of actions, procedures and responses needed to realize a promise might change from moment to moment, if the environment changes.*

Now, it is true that computers cannot really promise anything. They do not think and they do not have "free will". The primus motor for machine promises ultimately comes from humans who control them. But in practice, we program computers to try to fulfill certain promises that humans decide. For that reason, it is convenient here to abstract away these unimportant differences and to speak of machines as if they could make decisions for themselves. All this means is that decisions can be (and often are) made separately for different computers as if they had made their own decisions. So why bother?

We need to be able to talk about things that are planned for, things that are purposely avoided, as well as things that actually happen. And in fact all of these things can be discussed in a single framework of promises, and the probability with which they are going to be kept.

Management is a fundamentally abstract business: it is not about what happened, but about what might be. And while a small 'folksy' nay-sayer minority will always decry the suggestion that an abstract idea could contribute usefully about the real world, we must recognize that no field of study comes of age until it can abstract general principles and frameworks for thinking.

Promises are much more than a theory of events, or a model for planning, they describe long-term, on-going interactions, they are fundamentally service oriented but they go far beyond the Service Oriented Architecture. They have a business flavour to them. They are intuitively something we know about.

When we fully understand the implications of promise theory, we shall have easy tools for management that allow us to surf the ruts of technical detail, or dive into it at our option.

I place promises on a pedestal as my number one, Single Most Important Idea To Grasp because they are atomic units of *policy*, *management* and most importantly of all describe *voluntary cooperation*. Promises are the key to describing our lack of control over communities and assemblies of components working together. They are a way of admitting to our most basic limitations and working around them. This advances a principle of scientific honesty over wishful thinking.

2 The point of the arrow is not to oblige

Why voluntary cooperation? Why not simply force people and machines to cooperate (i.e. oblige them)? There are two reasons why the notion of promises is superior that of obligation:

- Obligations are a problematic ideal. Obligation is never entirely enforcable. Obliging someone to empty a lake with a sieve is not going to get us far. Obliging a server to give a client access to a resource, sounds more like an attack on its integrity than a policy decision by the server.
- Obligations oversimplify the information 'dilemma' in decision-making. Obligations on an unsuspecting party can originate from many sources and so an agent does not always have direct knowledge of what is being required of it.

The obligation model, for all its historical precedence, is simply the wrong way model cooperative behaviour. Promises, on the other hand, represent a polar extreme of viewpoint from which all other viewpoints can be constructed, insofar as they are realizable at all. It represents an *atomic theory* of computer architecture, and *promise types* form its table of elements, as we'll see below.

Promises immediately 'solve' the philosophical problems with obligations because they remove the assumption of absolute guarantee, and they make every single agent responsible for themselves. If someone promises to empty a lake with a sieve, the recipient of the promise has the responsibility to judge whether it is likely that the promise will be kept: it takes the risk, which is much more reasonable than pretending to oblige the impossible. The model forces each agent individually to confront the realistic uncertainties that are inherent in distributed cooperation.

Moreover, the agent who will have to complete the task is the only one who has the relevant information about whether it can be done. If we do away with obligations, then all of the information required for making and completing a promise is localized within the promising agent.

> *Obligations tend to distribute implementation information and requirements, whereas promises tend to localize all relevant information.*

Let's recap, we have to grasp a basic reality: we can never guarantee what will actually happen in any system, we can only express our intentions and hope that all of the things we were not able to take into account will not ruin our plans. A promise is more than an intention: it implies a best effort, but offers no guarantees. Nothing is omnipotent, but anyone can offer a service. Computer modellers are used to assuming that everything they decide will come true because they have traditionally been able to build local computer architectures with execution privileges where this could be approximated using suitable error correction techniques. In system administration we do not have the luxury of this convenient illusion.

3 Some vocabulary and some philosophy

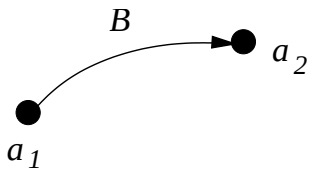
Let's write down promises more formally and develop a little philosophy to make sure that the idea of promises agrees sufficiently with our everyday understanding of the term. We need to start with a set of 'agents', i.e. people, computers, components etc. which are the elements of a system capable of making promises either implicitly or explicitly. Let's call these $a_1, a_2, \ldots$ etc. We call these agents autonomous, meaning that each agent can freely reject any impulse or suggestion from another agent. No agent can be forced or controlled against its "will".

Note that we can think of something as mindless as a computer program to be an agent, since it makes certain promises in terms of the functionality it offers. We don't bother ourselves with the problem of whether it has sufficient sentience to be able to make autonomous decisions, because this is just a simple model (an abstraction). *Someone* imparted that intelligence on its behalf, and that is enough. We use as many or as few agents as we see fit to model a given situation; usually, the more the better. We shall see the virtue of this below.

Next we shall need some arrows. A promise of B from a_1 to a_2 is written:

$$a_1 \xrightarrow{B} a_2.$$

which, of course, offers a simple picture, the beginnings of a network:



and we shall prefer this graphical form over the mathematical expression in almost all cases, especially as the number and cardinality of the promises increases. The *promise body B* is a description of what the promise is about. I think it should contain two things: a *promise type*, which explains the subject of the promise, and a *constraint* or *variation*, which describes the extent of what is being promised. An agent can make promises of different types, e.g. promise a rose garden, or promise to pay a bill. Similarly each promise of a given type has a constraint, e.g. a rose garden of a certain size or colour, or a bill of up to a maximum amount.

We assume for simplicity that the promise types do not overlap with one another (this is just good housekeeping). So having promise types of a rose garden and a rose bush would be asking for a mess (since a rose bush can be thought of as a small rose garden). There is no problem if an agent makes two promises of different types to another agent, e.g. promising a rose garden and promising to pay a bill. Nor is there a problem in making two completely identical promises, e.g. I promise to catch you if you fall and I promise to catch you if you fall (this is the same thing). However, we shall abhor making two promises of the same type with different constraints, e.g. I promise you will receive a response from the web server within 2ms, and I promise you will get a response within 4ms. This is a contradiction, or we shall say *broken promise*.

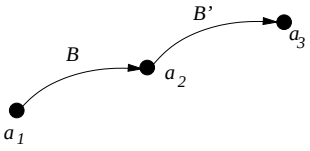
Now, some people might argue (and you know, this means they already have and therefore I'm going to kill this before it gets out of hand) that it would be okay to promise a server response within 2ms if you had already promised in 4ms, since 2ms is better than 4ms and certainly falls within the bounds. But this opens up a Pandora's box of problems, not least of which is a promise ordering ambiguity which I know is going to get us into trouble. Besides, this is based on a prejudice: who are we to say that 2ms is better than 4ms? That is a subjective judgement and it doesn't work for rose gardens. If I ask for a bush and get a garden (or vice versa), I might not be happy. Perhaps there was important reason to ask for exactly what was asked. So let's not even open that box. This is why theories use models: we make up rules to avoid obvious traps that less considered folks might fall into.

So, to summarize, repeating the same promise twice is okay. Promises are *idempotent*: repetition confirms but they don't add up cumulatively. A contradictory promise, on the other hand, is at best muddying the waters, so we simply say that this is bad and call it a broken promise. In principle, we might like to disallow broken promises, but we've already said that we cannot force agents to do anything, so we can't stop them either. We know from experience that people make all kinds of contradictory promises, sometimes to obfuscate but often out of ignorance, and this is part of the problem that promise theory will help us to solve. So we recognize this as bad, but we should not make the mistake of ignoring it. Another rule to keep the theory simple:

> *The promise body cannot explicitly refer to the identities of agents. Only the arrow can route promises.*

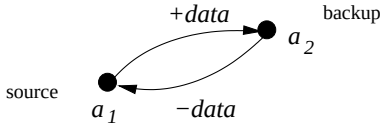
As with any system, the rules we invent allow us to build 'valid' and 'invalid' worlds from the building blocks. These rules are very lightweight compared to many formal logics: the rules will only affect the very highest meta-levels of cooperation within a system, but these are the levels at which management often goes awry. Detail technicians do not often make mistakes, but planning errors or architecture misunderstandings are two a penny.

Promises are not *transitive* in general. If a_1 promises B to a_2 and a_2 promises B' to a_3, then it is not true that a_1 has promised anything whatsoever to a_3. The graph looks like this:



but we should not make the mistake of assuming that promises are like a flow of communication. Promises are not like routing (although routing involves many promises between autonomous nodes).

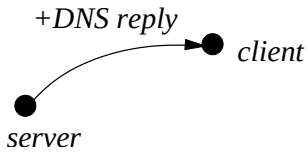
Last but by no means least, because promises anchor a sender to a receiver with a directional arrow, they form *networks* and hence there are two kinds of promises: promises to give and to receive. We write these using a $+$ (give) and a $-$ (receive) sign. For example, to ensure a remote file copy, such as data backup we would need two promises:



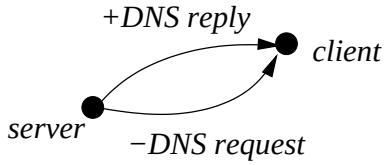
Here, the source of the data a_1 promises to make the data available to a_2 (perhaps by running an appropriate daemon and granting some access controls (for a cfengine-like download) or by starting an rdist-like "push" of data to the receiver). Note, however, that this does not (and cannot) oblige the backup agent a_2 to download anything or receive the data. It must, of its own 'free will' promise to receive the data. This could mean instigating a network copy (if the copy is based on a "pull" model), or opening appropriate access controls (if it is based on a "push" model). The promises are agnostic towards the implementation details, but they encapsulate the features of the interaction.

4 Example

Let's see how promise theory helps us to understand something we think is obvious: the Domain Name Service. Given that promises are service oriented, we might think that it is trivial to model this service by a single promise from a server to each client. The promise made by the DNS server to reply to requests is certainly important:



The client, after all, makes no promise to send requests to the server. It might, it might not, but there is no promise involved. The server, on the other hand, makes a general promise (usually to all agents with IP addresses) to provide information. This, in fact, involves two promises:



Here we have a promise to receive the requests from clients and a promise to provide data in reply. Again, this is an entirely one-sided affair. This might make us suspicious: what do these promises correspond to? Does the client have no responsibilities in this matter?

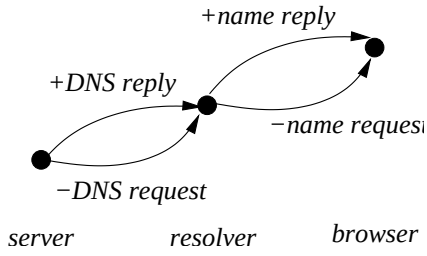
Doesn't the client have to send something in order to get a reply? Yes, but forget about that. When you think at this level, you are thinking about a

communication protocol, not about promises. In promise theory we are way above that level of thinking. A promise is not a message. The promise made by the server to reply is not the reply itself, it goes beyond any single reply. It is a responsibility to be fulfilled. In fact, Jan Bergstra of the University of Amsterdam and I have decided to use the term *commitment* for a promise that requires a non-returnable investment of resources on the part of the promiser. That name applies here, since a DNS server undergoes some costly load in performing its function.

Turning high level promises into low level details is part of a discipline that helps us to understand systems from the top down. Conversely, you can see how to replace a high level promise with many more specific low level promises, or build high level ones from low level ones bottom-up. At the management level, we are interested in the promises, but at the technical level we need to know how to implement them. In this case, the answer is quite easy: a promise by the server to use data from the client can be implemented by making sure that there is a server process with an open port 53 and appropriate firewall access rules to admit requests from the client receiving the promise (usually this is everyone, but it need not be). The promise to provide answers is also covered by running the daemon and having updated zone files containing the data.

> *Breaking a promise up into sub-parts is like procedural decomposition in programming, but it happens at the level of goals, not procedural recipes.*

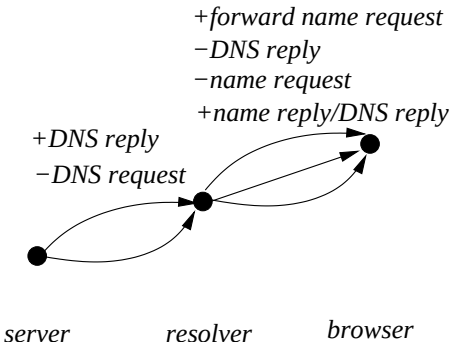
The client, sure enough, has no promises to make to the server, but let us suppose that we look at the client more realistically. What really happens is that we have a client program (say a web browser) that needs to look up data in the name service. There is an interloper: a local name resolver that promises name resolution locally:



Now it is the web browser that makes no promises, but the local resolver promises to give some answer to the web browser and to accept its requests. In

all modern operating systems, this promise is guaranteed by the kernel architecture so we do not have to do anything special here. This already indicates a number of things that can go wrong with the DNS system. Every promise is a potential cause of failure, but let's not get ahead of ourselves: we are not done, because the system does not yet work.

There is no automatic connection in the promise graph above between the promise of name resolution and a promise of DNS resolution. The resolver can fulfill its promise by saying "I don't know" to every request. It agreed to heed the request, it agreed to reply, but it did not agree to forward the request to an authoritative database in order to relay its response. We need two more promises:



i.e. a promise by the resolver to the client forward the request to a known DNS server, and a promise from the resolver to the client to use the reply it gets from that source. This promise can be implemented for instance by configuring the Solaris `nsswitch.conf` file or equivalent.

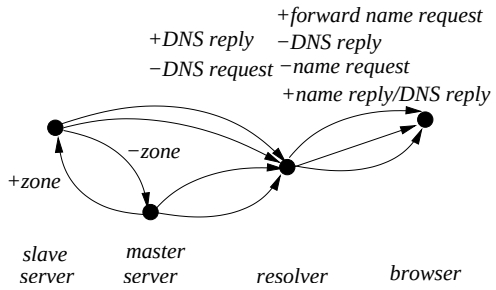
Notice that the guarantees are directed towards the party that is interested in them. If we were thinking communications bindings we might think that the promise to use or receive the DNS reply would go to the agent that sent it (the DNS server), but the server doesn't care whether we use the reply or not. It has discharged its responsibilities by simply replying. What the promise says is that the resolver should not merely ignore the reply but use it in formulating a reply to the client.

You might think that I am being deliberately obtuse here, and you would be right. In order to understand how independent (autonomous) components need to be bound together in cooperation, we, must follow this discipline of breaking things down into interactions between dumb components and their promises. You are probably surprised at how complicated the DNS system actually is. We tend to think of it just as a simple server process, or a `resolv.conf`

file, but these are just a couple of the requirements to fulfill the promises above. Moreover, we've only just started to model the DNS system. What about the promises that are needed to model master and slave servers?

Currently the client in this example does not promise anything, because it is only a casual end-user. DNS service provision is not contingent on what the client wants. The matter is different however if we consider the interaction between a master server and a slave server. There a master has to promise its data to the slave and the slave has to promise to use the data, as well as provide the same kind of service as the master. All this requires the coordination of even more promises.

When we add all of these things we end up with this picture



Now we start to see a number of things that can go wrong, but also a certain amount of redundancy in the graph. The promises allow us to see where things can go wrong if one of the promise arrows does not deliver on its promise.

The point is not that we could not have figured this out without the promises, but that we can start from the simplest drawings, without knowing the details of BIND or `gethostbyname()` or different operating systems etc.; we have predicted all of the potential problems by sketching out the main agents and by considering what kinds of promises are needed to get them working together.

5 The art of promising

Since I started to think about promises, I have noticed just how easily and casually people make promises that they have no chance of being able to fulfil. Given that we make promises so lightly, none of us should be surprised that systems involving human beings fail from time to time.

The example above shows how the discipline of decomposing into agents and promises allows us to see how a system works. Even in the simple example you probably saw something that you likely would have taken for granted,

even without getting "down dirty" into configuration files etc. One of the strengths of this way of thinking is that it does not let you take things for granted.

This is a useful planning discipline: each time you draw a component, or agent, you should ask: why should I expect this agent to behave as I want it to? Then you should turn that desired behaviour into a list of promises the agent needs to make to ensure the behaviour you want, bearing in mind that agents do know about each other's decisions or promises, unless they are a recipient of promises.

This way of thinking might seem like a small step, but its implications are huge: it is forcing us to ask what are responsibilities of each component of the system? We must confront every expectation of behaviour, not just the ones that we think are big and important. Not just the components for which there is an O'Reilly book. The entire chain of responsibility begins to appear as we start thinking in these terms. Of course, we shall still want to suppress many of the details. Every promise that we need to make a distributed system work is a promise that can potentially be unreliable. It is a potential mode of failure. This allows us to plan redundancy.

But there is something else to be said about promises: they allow us to understand the real source of fatal attraction between independent decision-making bodies: commerce.

6 Promise of riches

People, systems, or businesses work together only when it is beneficial to them to do so. There is profit to be had in delegating (or "outsourcing") to a specialist or building on others' successes (like the DNS).

In fact, each agent can attach a value to the promises it either makes or receives. An exchange of promises then becomes a currency for trade. "I promise you X if you promise me Y", and vice versa. No dollar payment necessarily has to take place (though one can easily include money in the battery of possible promises). One could trade access to web services for an accurate time service, or a hotel could trade network service for all guests for cheaper rooms for its employees. A whole economic network of commerce and cooperation can be understood in terms of the promises that are made.

When promises do not go in both directions, a relationship is in danger of falling apart and we should be suspicious of single promises that are not balanced by a counter-promise. It is a sign of potential instability.

For example, in a normal trade interaction one kind of goods is exchanged for another in such a way that the individual agents' personal valuations of the goods measure the values to be equal. This exchange is easy to understand. If one party stopped promising goods, the other would soon stop too.

A less obvious example is what happens when we recycle waste. Users of glass, paper, metals etc promise the collection company to separate these materials into appropriate containers to be collected by an appropriate company. The company promises us that it will remove the used materials and do something productive with them. The value to us:

$$V_{\text{user}}\left(\text{user} \overset{separate}{\rightarrow} \text{collector}\right)$$

of that promise to separate the goods is negative – it costs us some effort to do this, so it is unclear at this stage what we are getting out of this deal. The value of this promise to the collector, on the other hand:

$$V_{\text{collector}}\left(\text{user} \overset{separate}{\rightarrow} \text{collector}\right)$$

is a saving. This separation of materials will dramatically reduce their costs in reclaiming materials from waste and this could be the difference between making a profit and not. What about their promise to us?

$$V_{\text{user}}\left(\text{collector} \overset{collect}{\rightarrow} \text{user}\right)$$

The fact that they collect our waste is valuable to us. If they didn't we would have to pay someone to do the job, or do it ourselves at much greater cost. This much is obvious. There is also some moral value to this, not hard cash but a feeling that we are doing something good. Such abstract currencies should not be underestimated where humans are involved. And finally, the value of the promise to them is the recycled value of the materials themselves:

$$V_{\text{collector}}\left(\text{collector} \overset{collect}{\rightarrow} \text{user}\right)$$

It is easy to see that there is a flaw in this model. Suppose users do not bother to separate their waste, what is going to happen? The garbage collectors do not inspect our bins for transgressions and refuse to take the garbage if we

don't separate, so the economic threat of non-compliance with our promise to separate is not that strong. In practice most people do this out of a sense of moral gain or civic duty. This is a fine example of voluntary cooperation for abstract currency, and it is an effect that is more common than hard-line economists might think. But we see a vulnerability in the thinking.

Why are we looking at promises? Why not goods or services exchanged? Well, business plans are based on the potential value of promises, not on actual transactions. Promises do not necessarily go away when they are fulfilled, e.g. when someone collects the trash. Promises persist and what is interesting to us is how *stable cooperation* can arise and remain. Promises allow us to discuss the value of stable and persistent (business) arrangements, not merely to address fire-fighting measures.

The stock market itself also values shares and even currencies on the promise of companies and economies rather than their physical assets. It is all about how much someone might promise to pay (hypothetically). It is all abstract. Businesses do this kind of thing all the time, of course, but is this any kind of scheme for computers?

We already have one premier example of voluntary cooperation between autonomous systems (at a scale larger than businesses) and that is *peering* between service providers, i.e. mutual promises to transport packets between each others' networks. This is not a traditional computer science relationship of event handling. Indeed, recent studies showed that the value to companies from peering is less to do with the dollar value of the transmissions. It is the promises themselves that are considered to be of value, even if no packets are actually exchanged. It is a matter of being seen to be connected to the right people[2, 3].

> *Promise valuations answer the "why" question: why would the parts of a system choose to cooperate voluntarily?*

It turns out that traditional economic models, like the Principal-Agent model of contract economics, can be formulated very easily in promise theory (see the coming Handbook of Network and System Administration, eds. J. Bergstra and M. Burgess, Elsevier 2007). This economic interpretation, based on a common currency, has led some reviewers of my work to claim that:

- Promise theory is just game theory.
- Promise theory is just the Service Oriented Architecture.

Both of these claims are false, but both are rooted in correct observations: namely that promises combined with agent valuations lead to competitive game scenarios in some cases; and the service-like nature of a promise (the promise of service) between agents is superficially like interacting web services, though far more general.

Certain constellations of promises lead to economic games, and promising certainly involves service oriented thinking when realizing promises. But promises can exist and be analyzed before anyone has begun to fulfill them, as a management planning exercise. All you need is a pen, some paper, and a thinking cap. Sometimes the promise of something is enough to make something actually happen, e.g. in peering agreements. So it is more than the concrete realization of services through the web in SOA. Moreover, in SOA you might remember to model the client and server actions, but you would not consider to model the access control to that service as part if the model (the "receive" promise). Thus you would still be leaving out essential pieces of the puzzle.

7 Pansies in the rose garden?

People who have not grasped the essential point being made have said to me: why would you want to do away with the certainty of control and replace it by this wishy-washy federated pansy-boy nonsense?

This is missing a point. I am not necessarily advocating this as a solution (such things are a matter for policy decision, although wherever humans are involved human-rights groups tend to prefer this kind of thing), I am saying that it is inevitable. We are already there. The so-called "certainty" of "control" that we have partially enjoyed in the past was only an illusion that worked under very special circumstances, and those circumstances are rapidly evaporating in the environment of the Internet.

To effectively remove any doubt about cooperation, to impose laws etc, society must be able to retain the option to wield omnipotent, irresistible force against potential transgressors. Guess what, we can't. Military dictatorships do this to a reasonable approximation, but still imperfectly, as dissidents prove. The current situation in Afghanistan and Iraq shows how fragile our taken for granted lawfulness is. What we can try to do is business, commerce.

You can huff and puff or pray or beg, or jump up and down, shoot (only in Texas) or kick your servers, and even crash a plane into a building, but none of these things will force the electricity to come out of a broken power line. We

cannot oblige the electricity company to prevent this from happening: it is not possible. The electricity company can promise to do its best. And that is as good as it gets.

Well, you say, this is an extreme example to be sure. We can certainly force people to pay their bills. You can't. At best you can ask the police to intervene, or try to sue them, each of which are voluntary acts by you which in no way compels them. These threats might help to persuade them, but that is an entirely different thing. Without their cooperation, you can only try to punish their non-compliance.

The mistake people make is in assuming a causal connection between event and effect, when none is present. Someone can pay a bill without ever having used a service, or never pay in spite of using a service. There is absolutely nothing linking these actions other than *voluntary cooperation*. The fact that service and remuneration promises are strongly *correlated*, i.e. that they usually occur together is what makes the collaboration a potential economic success: "when you do X, I'll do Y". There is thus a kind of natural selection taking place, of an economic variety.

I ask you, as a reader, to stop and think about this, because if you don't "get" this simple idea, that we are essentially powerless to force anyone to do anything, then the rest of this diatribe will be lost on you and you will be stuck in the mythological Control Age.

At best we can put together an irresistible portfolio of services and incentives to coax autonomous parts of a system to work together. This could be a complicated matter, understanding what makes people tick. Welcome to the world of commerce. Understanding how this works requires *network science*, it requires us to know how relationships are held together in a web of interactions. Promise theory encourages us to draw this web as a network of promises, and then to attach valuations of each promise by each agent to understand their perceived gains or losses.

Today, much business is carried out by voluntary cooperation and decentralized operation. The main tool for this is the Service Level Agreement (SLA). This is a document that essentially contains bundles of mutually agreeable promises by two parties. To make the agreement, both parties then promise to honour their respective promise bundles. The deal is easily expressed in terms of promises between the two parties, and potential problems or broken promises. This is not a nice predictable, controllable regime; it is a mess, but a self-sustaining mess.

So, I beg your pardon, I didn't put the thorns and the weeds in the rose

garden. They are just getting harder to control, and that means we need to understand the real ecological problem of system management. Kings and dictators are out of fashion. Blame it on the Internet.

8 Pointing to a new direction

When I started writing cfengine, no one had much of an idea about configuration management or how the world would look in the coming years. I drew my inspiration from a combination of a federated model, existing tool syntax and an intuition about how a declarative syntax could sieve out the golden rules from the irrelevant gravel of loops and decisions within imperative scripting languages. The initial effort was about user-friendliness, i.e. simplifying the appearance of the problem. However, as the years progressed, it became clear that there was a need to build in safety features and principles of operation to avoid problems and have guarantees.

Later, dissatisfied with the ad hoc discussions and opinions surrounding cfengine and other configuration management tools, I wanted a language for reasoning about systems in a way that would include the essential principles that cfengine endorses, as all previous modelling techniques make assumptions about obligation that cfengine refuses to make. Only then would we be able to discover whether a given configuration management system was right/wrong, better/worse, to be able to compare different ideas and find faults and weaknesses in the existing implementations.

I assumed that this language, whatever it was, would give some insight into how I could tidy up the evolutionary mess in cfengine's language interface, without sacrificing the principles that have made it relatively safe and reliable over the years. Little did I know that promise theory would be good enough to define it completely, and that it would unify many management different issues under a common framework.

Promises are a goal based approach to modelling, mostly order independent, which fits cfengine's goal based approach to configuration, but it also offers some insight into how we can understand sequential changes. Even imperative, sequences of actions are compositions of many intermediate goals, hence we can also represent time ordered sequences in this way too. Conversely, unnecessary time ordering is suppressed.

At about the same time that I was developing cfengine, two researchers Marriot and Sloman were pursuing a related idea at Imperial College in England: policy based management, a term I later came to use in deference to them.

Whereas I started with a prototype 'autonomic' system, they developed ideas first and only later began to implement a kind of language for policy based access control. They have explored many alternatives that are unlike promise theory. How do promises relate to these? Most of them are about obligation.

Modal logic was one possibility that was explored as a language. There are many kinds of logic. Deontic logic is the logic of obligations and has been proposed for that reason, but only with reservations as it has many problems. I have already explained why obligations are bad.

In case you are thinking that every promise implies an obligation, let me persuade you otherwise. An obligation means that you are not free to withdraw a promise once made (an obligation is half of a threat). A promise is a unit of voluntary cooperation.

9 Aspects, bundles, genes and ontologies

No, not oncology nor odontology. Ontology is a form of knowledge classification and management, which is becoming important for mapping out information systems including policy and requirement engineering. The designers of the Semantic Web are studying this, for instance.

It is also something for promise theory. The reason is straightforward. How shall we decide on a lingua franca for promises, i.e. the list of promise types and their parameters? This is somewhat analogous to looking for the genetic code, (which represents a set of promises for making protein parts).

Alva Couch[4] and I proposed recently an approach to management building on the idea of promises. Because promises are very low level objects (remember they form our most elemental, atomic theory of management) it could be hard to understand higher level concerns in terms of promises without a way of organizing them, just as it would be difficult to understand the functioning of a motor car at the level of atoms, or an eyeball at the level of genes. Aspects are a way of organizing promises into distributed bundles and constellation.

The philosophy of *aspects* has entered into computing in several ways over recent years. The idea is simple and has to do with convenient *separation of concerns*. It is universally agreed that the ability to keep logically separate issues separate is a desirable goal in organizing systems. The trouble is that it is not always clear how this can be done, especially when other philosophies are in play. Object orientation, for instance, can propose one model for organizing a computer program, only to find that certain issues "cross-cut" the OO class

model. For example, suppose you built a number of class containers to manage Text, Image and Style Sheet objects. All of these potentially contain colour information. An aspect of web pages that affects all of these is "colour contrast". If we wanted to increase or decrease the contrast on a web page this would require coordinated changes in all of these 'separate' classes for text, images and style. Contrast is therefore an aspect that is distributed across several OO classes.

In a similar way, aspects that affect the functioning of a collaborative enterprise can be distributed across many agents. These agents will then have to promise to collaborate, and share relevant information, if they are to implement the collective wish to coordinate their 'contrast' or whatever property we are interested in. This is how businesses work, as long as it is economically viable.

If we are astute we choose promises so that they do not have overlapping concerns, because then we would be able to compile any kind of aspect of system management down into a number of cleanly defined, obviously non-contradictory promises. But a promise to have a certain colour value can appear in any number of separate promise bundles (representing text, images etc), thus aspects are also simply bundles that cross-cut others. Bundles can overlap, but not atomic promises.

If two promise types could overlap there is clearly the danger that one could end up with broken promises in non-obvious ways (making management harder), so as long as we can find a clean *ontology* of basic promise types to cover the whole spectrum of aspects we need to govern, then it will be possible to find errors of design trivially by counting promises of a certain type. Imagine how awkward it would be if we had genes that switched on contradictory processes in our bodies. Such genes would not survive long in the selection pool.

Coming up with such a taxonomy of promise types is a challenge in its own right. Promise types are not uniquely definable. Anyone could come up with their own set of "genes" and build a whole new chemistry of cooperation. This is why we need ontologies to understand these different chemistries. Ontologies will bring back logic into the set of tools for designing management systems, but promises themselves are graphs and there is also logic in the composition of the graph. Ultimately, we see how promise theory can meld the three essential ingredients of a stable network of cooperative behaviour: *who*, *what* and *why*.

10 A rose garden perhaps

Sometimes it takes years to make the crucial observation that reveals the simple and obvious essence of something. I believe that promise theory has this potential for all kinds of management. It is both simple, easy to grasp and powerful. With a few tools, it could be a powerful modelling framework. I am so convinced that I am basing cfengine 3 entirely on this model.

Apart from being a promising theory, not least for pun-loving archers, it is an easy theory to understand. You don't need a degree in mathematics to be able to get something out of it. Conversely, with a degree in mathematics, you might be able to make some exciting discoveries. As an intuition aid, it is helpful. Moreover, the principles unveil a simplicity behind the potential complexity of network activity, provided you don't violate the rules of autonomy. When people make mistakes thinking about promise theory, it is usually in assuming the promises are obligations. This seems to be a hard habit to break.

Complexity in management arises from networking effects, just as the complexity of biology is in networking not genes. It is the ecological (economic) thinking that is truly challenging. Promises help us to unravel this by going back to primitive, elemental assumptions. Perhaps DNA networks too can be thought of in terms of the promises they make. Genes give us different promise types, alleles give different promise variations, and the networks of cells making such promises form organisms. What promises are necessary to make self-replicating systems and hence drive evolution?

Finding a simple theory is never easy. It is always easier to make something complicated. One of the strengths of the simple theory of promises is that it separates the identities of the agents from the promise bodies being made. This simple feature alone (which we can call a design feature of the model) avoids countless problems of recursive logic that have plagued previous attempts. If we can hold to that discipline, management does not have to be difficult.

The dialogue between theory and practice can now be drawn in pictures and made just as formal or as informal as suits your taste. If promises are adopted as a path towards getting system administration into a more engineering frame of mind, then I will be more than pleased.

And so they all promised to live cooperatively and economically ever after.

FOR FURTHER INFORMATION SEE HTTP://RESEARCH.IU.HIO.NO

References

[1] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97–108, 2005.

[2] W.B. Norton. The art of peering: The peering playbook. Technical report, Equinix.com, 2001.

[3] W.B. Norton. Internet service providers and peering. Technical report, Equinix.com, 2001.

[4] M. Burgess and A. Couch. Modelling next generation configuration management tools. In *Proceedings of the Twentieth Systems Administration Conference (LISA XX) (USENIX Association: Berkeley, CA)*, pages 131–147, 2006.